Course Project: Malware Attacks

Karson Hodge 9999-03235

Contents

Research Goal

Is attack type dependent on internet protocols, packet size, and/or duration.

Abstract

Malware attacks have serious implications and can cause millions and even billions of damages to companies. These attacks fall under several categories related to software security which include Distributed Denial of Service (DDoS), Denial of Service (DoS), Mirari, Recon, and Phishing. The goal for this paper is to examine elements that may inform security specialist of weaknesses or patterns that these attackers might utilize. Independent variables for this study will involve internet protocol, packet size and duration of attack and the dependent variable will be the attack type. The dataset used for this study will be Malware Attacks (Synthetic Dataset About Malware Attacks) created by renjiabarai. It was created using the Python Faker Library which generates fake data for analysis. There are 80,000 rows and 25 different columns for a total of 2,000,000 entries in the table. Columns are separated into four groups: Identical Info, Protocols, Flags, and Attack Type. A general explanation of each attack will also be given to highlight the effects and countermeasures that can be implemented.

Research Question

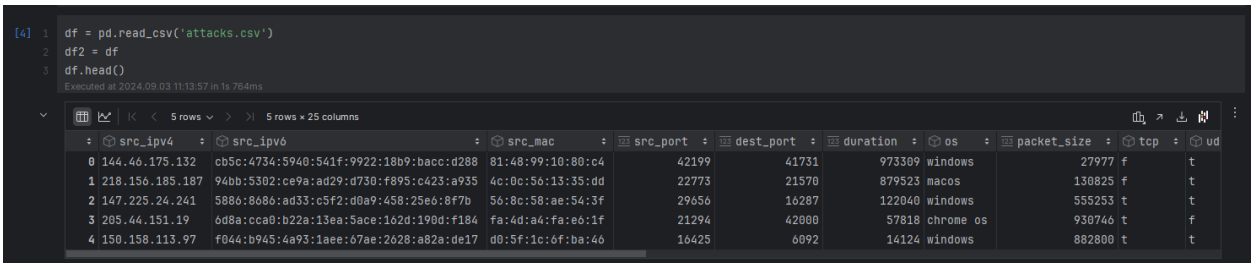
The following question is listed below and is the overall focus of the paper:

- Are there independent variables in the dataset that can determine what type of attack will occur or is it purely random by using classification and other machine learning techniques?

Introduction

An exploratory search will be the first step in demystifying the dataset and gaining an understanding of what columns and values are at our disposal. Preprocessing and normalizing the data is the first step in ensuring that the data is reliable and clean for our study. After, the data will be visualized to illustrate any visual patterns that may exist.

Exploratory



```
[4] 1 df = pd.read_csv('attacks.csv')
    2 df2 = df
    3 df.head()
```

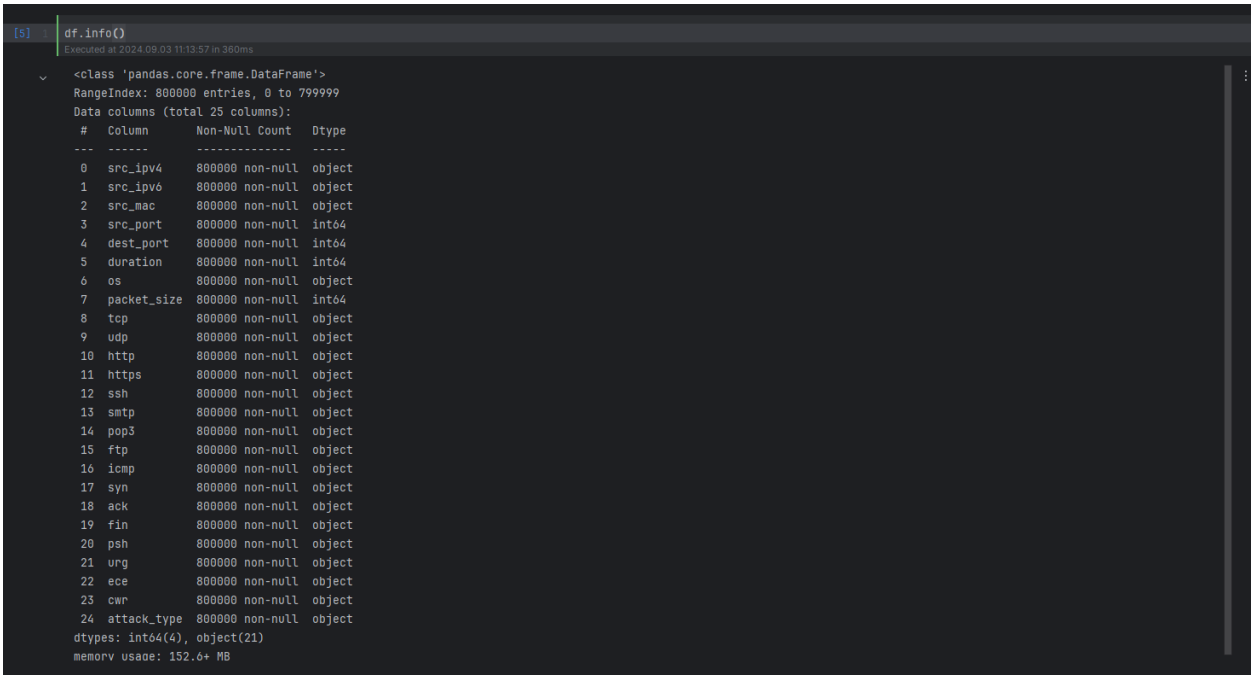
Executed at 2024.09.03 11:13:57 in 1s 764ms

5 rows × 25 columns

	src_ipv4	src_ipv6	src_mac	src_port	dest_port	duration	os	packet_size	tcp	ud
0	144.46.175.132	cb5c:4734:5940:541f:9922:18b9:bacc:d288	81:48:99:10:80:c4	42199	41731	973309	windows	27977	f	t
1	218.156.185.187	94bb:5302:ce9a:ad29:d730:f895:c423:a935	4c:0c:56:13:35:dd	22773	21570	879523	macos	130825	f	t
2	147.225.24.241	5886:8686:ad33:c5f2:d0a9:458:25e6:8f7b	56:8c:58:ae:54:3f	29656	16287	122040	windows	555253	t	t
3	205.44.151.19	6d8a:cca0:b22a:13ea:5ace:162d:190d:f184	fa:4d:a4:fa:e6:1f	21294	42000	57818	chrome os	930746	t	f
4	150.158.113.97	f044:b945:4a93:1aee:67ae:2628:a82a:de17	d0:5f:1c:6f:ba:46	16425	6092	14124	windows	882800	t	t

Code 1.

A copy of the dataset was created for later use in the analysis section. The original dataset will be used to display graphs and other charts using the original data. The head function is a great tool to view the first 5 rows of our dataset and get a feel of changes that need to be made.

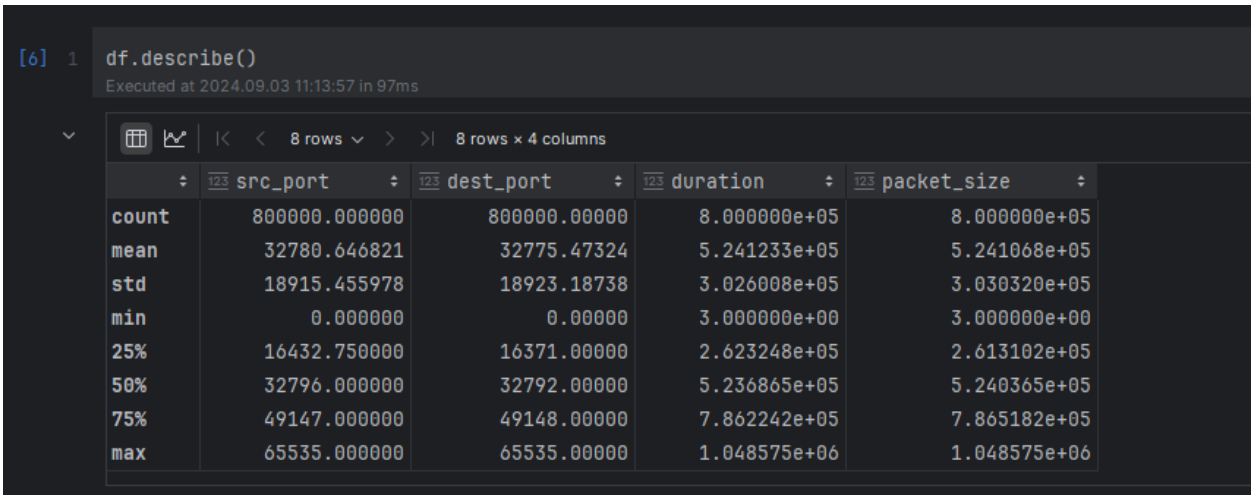


```
[5] 1 df.info()
Executed at 2024.09.03 11:13:57 in 360ms

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800000 entries, 0 to 799999
Data columns (total 25 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   src_ipv4     800000 non-null  object
 1   src_ipv6     800000 non-null  object
 2   src_mac      800000 non-null  object
 3   src_port     800000 non-null  int64
 4   dest_port    800000 non-null  int64
 5   duration     800000 non-null  int64
 6   os           800000 non-null  object
 7   packet_size  800000 non-null  int64
 8   tcp          800000 non-null  object
 9   udp          800000 non-null  object
 10  http         800000 non-null  object
 11  https        800000 non-null  object
 12  ssh          800000 non-null  object
 13  smtp         800000 non-null  object
 14  pop3         800000 non-null  object
 15  ftp          800000 non-null  object
 16  icmp         800000 non-null  object
 17  syn          800000 non-null  object
 18  ack          800000 non-null  object
 19  fin          800000 non-null  object
 20  psh          800000 non-null  object
 21  urg          800000 non-null  object
 22  ece          800000 non-null  object
 23  cwr          800000 non-null  object
 24  attack_type  800000 non-null  object
dtypes: int64(4), object(21)
memory usage: 152.6+ MB
```

Code 2.

The info() function is used to exhibit the characteristics of the dataframe. This includes column name, non-null values and datatype. Our data is a mixture of int64 and object.



```
[6] 1 df.describe()
Executed at 2024.09.03 11:13:57 in 97ms
```

8 rows | 8 rows × 4 columns

	src_port	dest_port	duration	packet_size
count	800000.000000	800000.00000	8.000000e+05	8.000000e+05
mean	32780.646821	32775.47324	5.241233e+05	5.241068e+05
std	18915.455978	18923.18738	3.026008e+05	3.030320e+05
min	0.000000	0.00000	3.000000e+00	3.000000e+00
25%	16432.750000	16371.00000	2.623248e+05	2.613102e+05
50%	32796.000000	32792.00000	5.236865e+05	5.240365e+05
75%	49147.000000	49148.00000	7.862242e+05	7.865182e+05
max	65535.000000	65535.00000	1.048575e+06	1.048575e+06

Code 3.

We use the describe function included with the pandas package to see the distribution of the data alongside the percentiles, max and min values.

Data Cleaning and Transformation

Duplicates

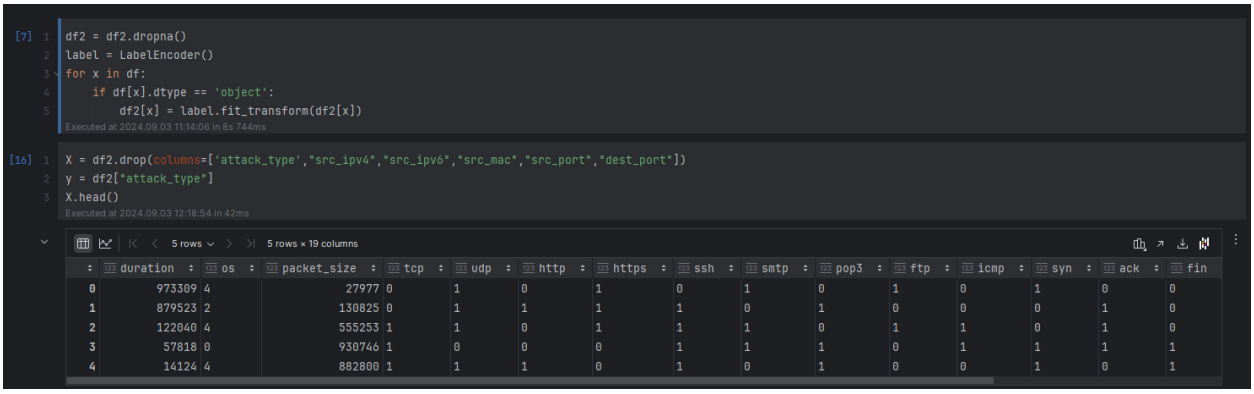
```
print(df2.size)
df2.drop_duplicates(inplace=True)
print(df2.size)
✓ [18] 470ms

20000000
20000000
```

Code 4.

The pandas drop_duplicates function was implemented to see if any duplicates existed. The size of the overall dataframe did not change so no duplicates were found. Inclusion of duplicates would have been an interesting choice in terms of preprocessing but alas none were found.

Non-Available and Label Encoding



```
[7]
df2 = df2.dropna()
label = LabelEncoder()
for x in df:
    if df[x].dtype == 'object':
        df2[x] = label.fit_transform(df2[x])
Executed at 2024.09.03 11:14:06 in 8s 744ms

[16]
X = df2.drop(columns=['attack_type','src_ipv4','src_ipv6','src_mac','src_port','dest_port'])
y = df2["attack_type"]
X.head()
Executed at 2024.09.03 12:18:54 in 42ms
```

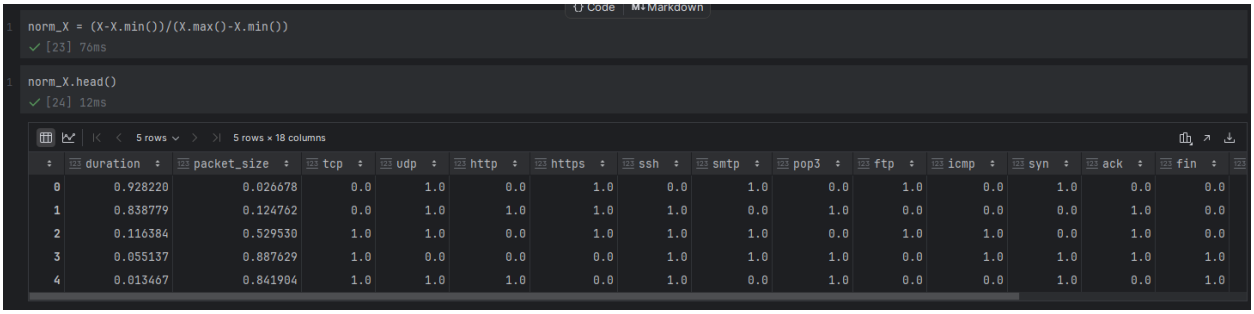
5 rows × 19 columns

	duration	os	packet_size	tcp	udp	http	https	ssh	smtp	pop3	ftp	icmp	syn	ack	fin
0	973309	4	27977	0	1	0	1	0	1	0	1	0	1	0	0
1	879523	2	130825	0	1	1	1	1	0	1	0	0	0	1	0
2	122040	4	555253	1	1	0	1	1	1	0	1	1	0	1	0
3	57818	0	930746	1	0	0	0	1	1	1	0	1	1	1	1
4	14124	4	882800	1	1	1	0	1	0	1	0	0	1	0	1

Code 6.

The internet protocol columns use true/false values for their entries, so for ease of use, these were converted to the int values 0/1 using the scikit-learn LabelEncoder function. A similar technique was applied to the attack_types column except encoding the values from 0-4. These transformations are performed for later analysis that will be accomplished in milestone 2 and 3. Before that, the dataframe was checked for any not available (na) values.

Normalization



```
norm_X = (X-X.min())/(X.max()-X.min())
✓ [23] 76ms

norm_X.head()
✓ [24] 12ms
```

5 rows × 18 columns

	duration	packet_size	tcp	udp	http	https	ssh	smtp	pop3	ftp	icmp	syn	ack	fin
0	0.928220	0.026678	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0
1	0.838779	0.124762	0.0	1.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0
2	0.116384	0.529530	1.0	1.0	0.0	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0
3	0.055137	0.887629	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0
4	0.013467	0.841904	1.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0

Code 7.

In code 7. We apply normalization to the dataframe so that a constant state is used instead of the different scales for each column. This will be helpful later when performing analysis between variables.

Visualization

Visualization will be performed with the non normalized and encoded values as for ease of understanding and scaling is not an issue at the moment.

Dependent Variable

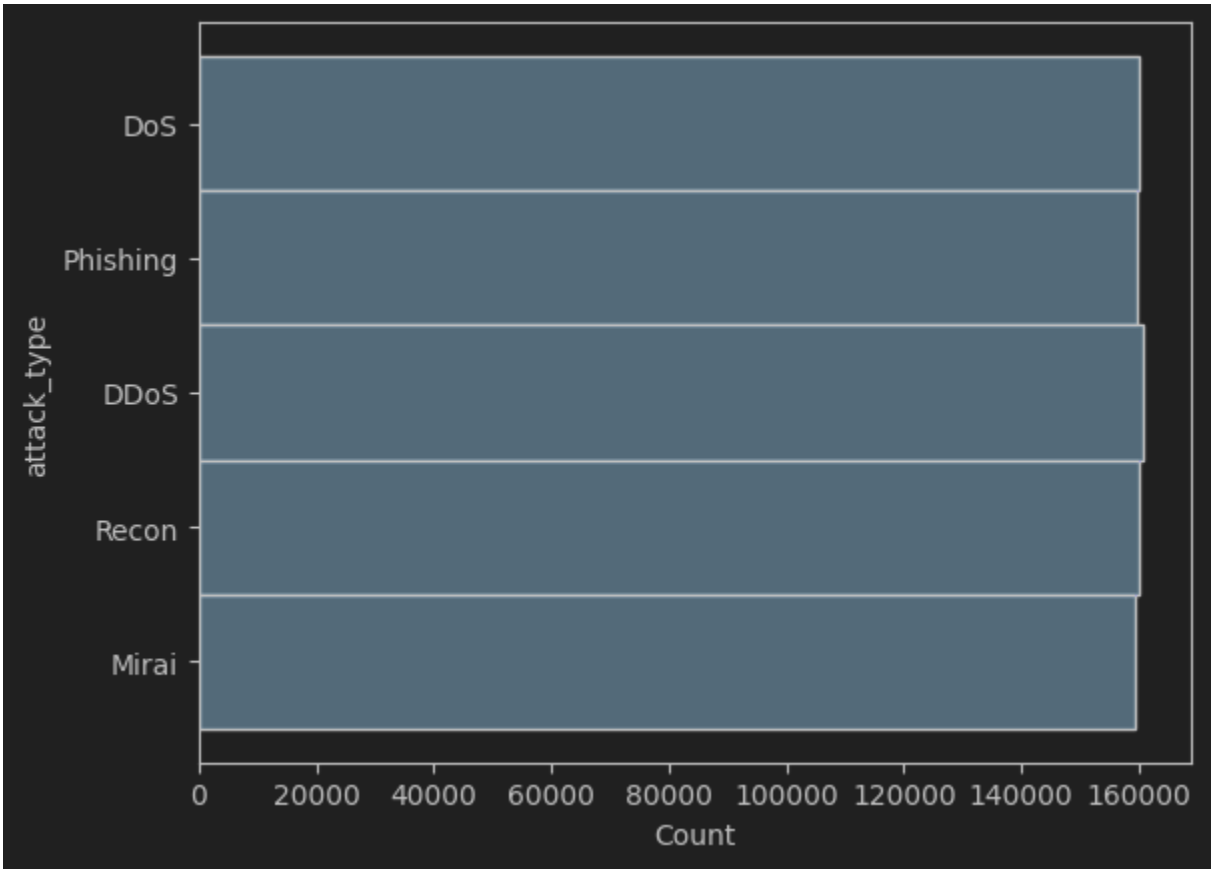


Fig 1.

We can see from Fig 1. that the attack_type column is uniformly distributed from the histogram plot. It can be inferred that randomly choosing an attack_type will result in a 1/5 chance for each option.

Independent Variables

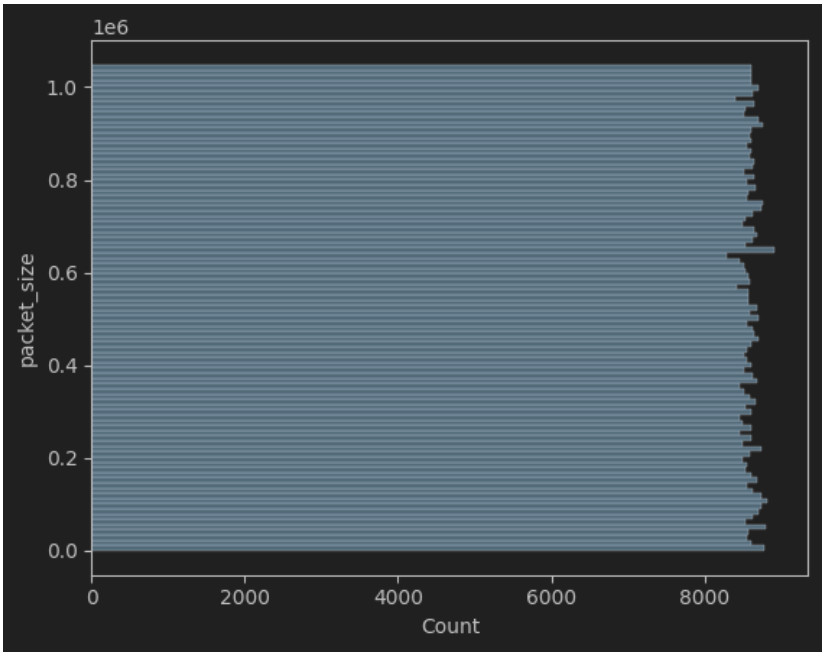


Fig 2.

Similar to fig 1. the histogram plot is closely related to the uniform distributed. And like attack_type, essentially every value has an equal chance of happening.

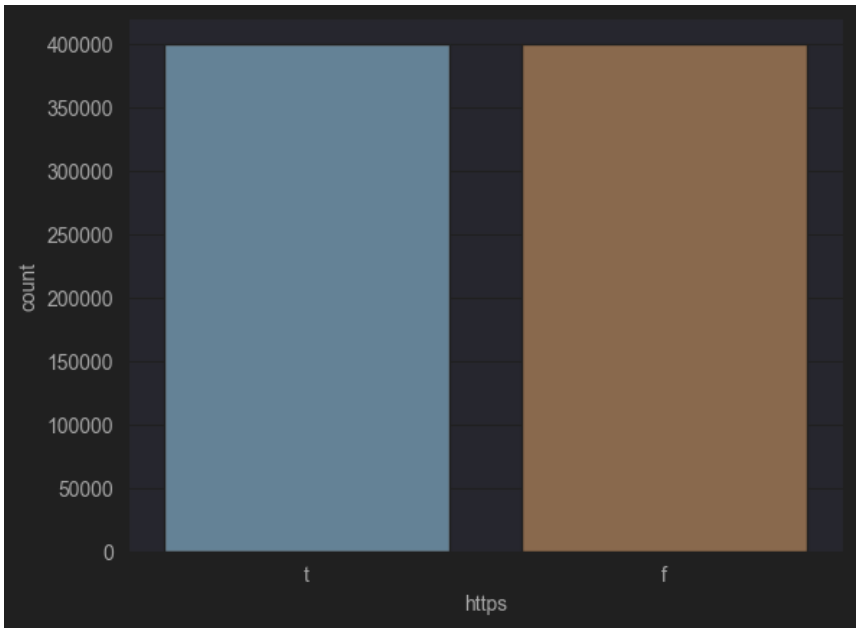


Fig 3.

Again, the generated data displays a relationship of 50/50 with no real variance between values. True or false can occur at an equal amount of ½.

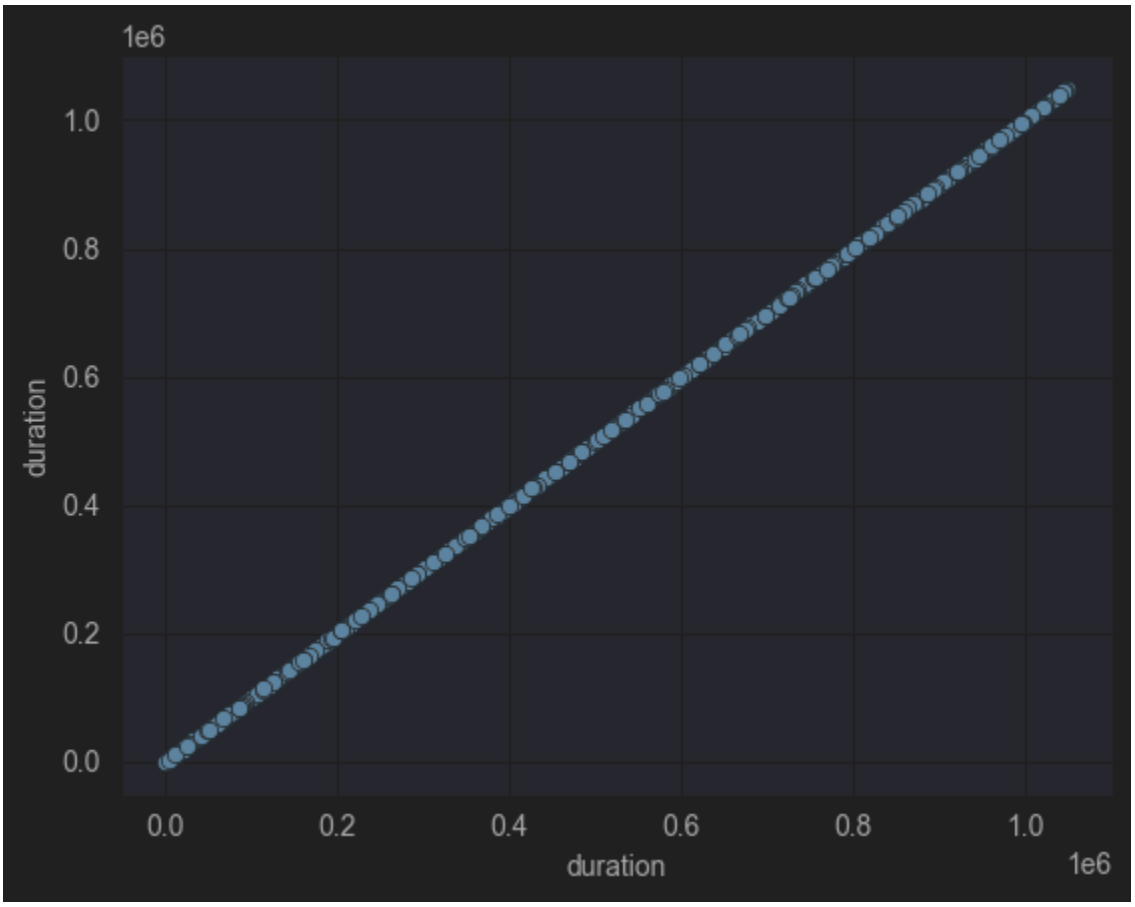


Fig 4.

For a change of pace with the overwhelming majority of histograms and bar plots, we used a scatterplot and plotted the duration against itself. The values range from [0, ~1.2e^6] and are evenly distributed down the line.

Related Contributions

The dataset on Kaggle has 3 directly related code sets to analyze, malware attack by Bosco J Garcia (https://www.kaggle.com/code/boscojgarcias/malwareattack), Malware Attack Classification by Sadik (https://www.kaggle.com/code/renjiabarai/malware-attack-

classification), and Malware Detection Hybrid by Md. Mithun Hossain (https://www.kaggle.com/code/mithun7755/malware-detection-hybrid). Two other code sets will have to be found to get the total up to 5.

We will first look at Garcia's codebase to determine what methods and techniques were used in his analysis. It appears that Garcia focused on similar independent variables as we did in our initial data cleaning and engineering. This required dropping 'src_ipv4', 'src_ipv6' and 'src_mac' from the dataframe. Label encoding was then used to transform the data into applicable int figures for analysis. The same process was applied to the dependent variable, attack type. All the data was split into testing and training sets that share a 30/70 ratio (test/training) and then fitted to a logistic regression model provided by scikit-learn. His end results were rather unsatisfactory with the model only predicting 20% correct guesses.

Fig 5.

The next Kaggle dataset we will examine is by username Sadic and was posted 4 months ago. Both the independent and dependent variables follow the same selection as Garcia's along with splitting the data into testing and training set (ratio 20/80). The main difference is the multiple models being used. 9 models were used to determine accuracy and precision, these models include CatBoost, AdaBoost, Decision Tree, Random Forest, LightGBM, Extra Tree, Logistic Regression, Ridge and Perceptron.

```
CatBoost
Accuracy Score: 0.1999875
Precision Score: 0.19995189891433207
***
AdaBoost
Accuracy Score: 0.19915
Precision Score: 0.19931201504965113
***
Decision Tree
Accuracy Score: 0.20113125
Precision Score: 0.20112960060050117
***
```

Fig 6.

Again, similar to Garcia's data, none of these models seem to surpass 20% in accuracy. This is valid because the data used is uniform and therefore the data will be 1/5 for attack type (1/5 = 20%).

Malware Detection Hybrid by Md. Mithun Hossain is the last of the analysis related to the chosen dataset. The main focus for Hossain's analysis is using Principal Component Analysis (PCA) to reduce the dimensions of the data while keeping information. Reduction of dimensions helps with the curse of dimensions and to prevent overfitting caused by the kitchen sink method.

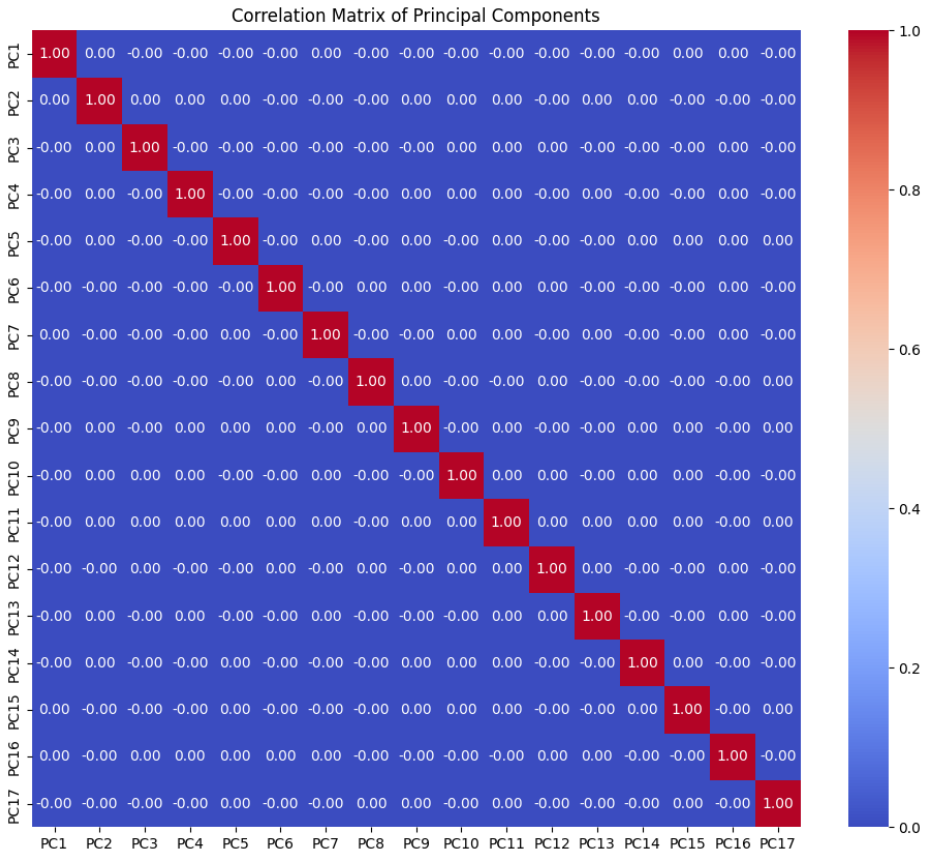


Fig 7.

The correlation matrix is used to find any patterns between the variables to help reduce the dimensionality. Since the correlation between variables is zero, a reduction cannot occur. Hossain plotted the distribution of these PCA variables to help the reader understand the distribution of the sample data.

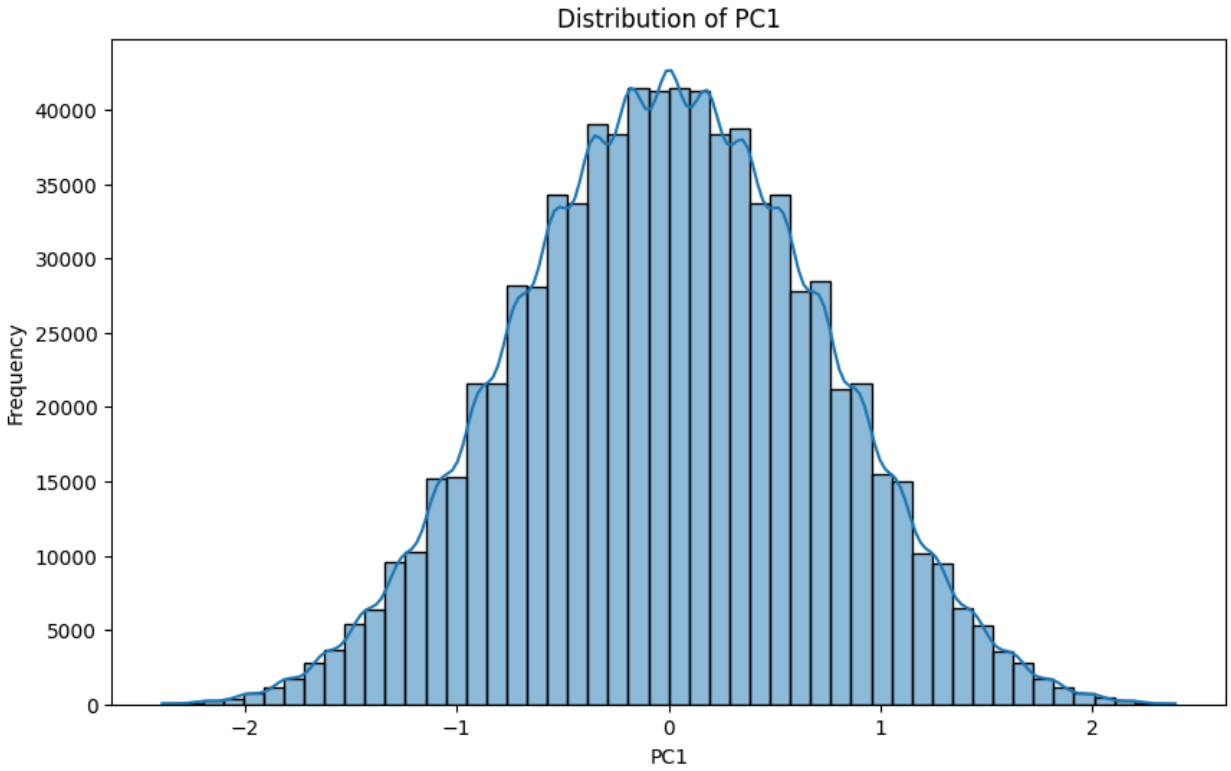


Fig 8.

The chosen dataset for this work only provided three code sets so a new set was found to supplement the remaining two code set analysis. Link to dataset: https://www.kaggle.com/datasets/luccagodoy/obfuscated-malware-memory-2022-cic.

After searching through the code tab, we settled on these two analyses. Malware Memory Analysis by Alper Karaca (https://www.kaggle.com/code/alperkaraca1/malware-memory-analysis) and ANN-based-approach for ML-based detection by Sripad (https://www.kaggle.com/code/sripadkarthik/ann-based-approach-for-ml-based-detection).

We will start with Alper Karaca's analysis of the data. The first step he implemented was visualizing the data to get a grasp of what it entails.

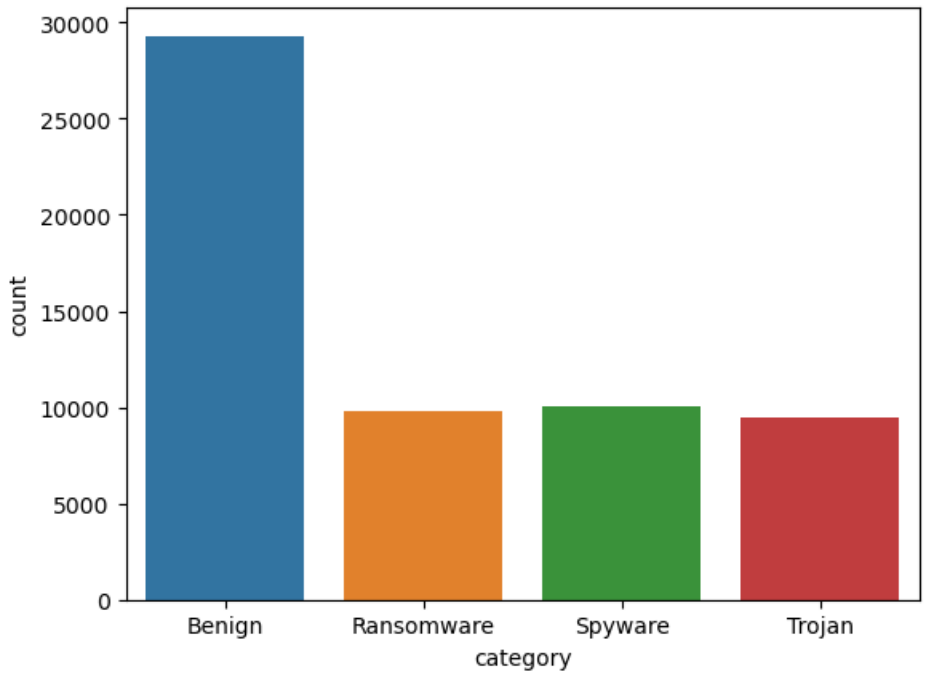

30000
25000
20000
15000
10000
5000
0
count
Benign
Ransomware
Spyware
Trojan
category

Fig 9.

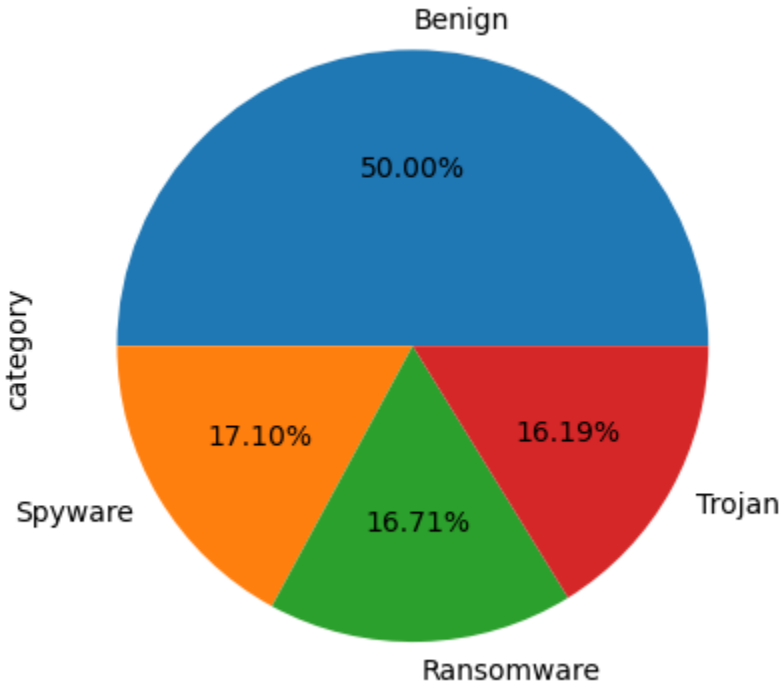

Benign
50.00%
category
17.10%
16.19%
Spyware
16.71%
Trojan
Ransomware

Fig 10.

His machine learning consisted of first separating the data into training and testing data and then applying the chosen algorithms.

	Model Name	Accuracy Score	F1 Score	Precision Score	Recall Score
0	Logistic Regression	0.999886	0.999887	1.0	0.999773
1	K-Nearest Neighbors	0.999886	0.999887	1.0	0.999773
2	Support Vector	1.000000	1.000000	1.0	1.000000
3	Decision Tree	1.000000	1.000000	1.0	1.000000
4	Random Forest	1.000000	1.000000	1.0	1.000000
5	Extreme Gradient Boosting	1.000000	1.000000	1.0	1.000000

Fig 12.

The 6 models trained include Logistic Regression, K-Nearest Neighbors, Support Vector, Decision Tree, Random Forest and Extreme Gradient Boosting. After performing the analysis, it was found that the Support Vector, Decision Tree, Random Forest and Extreme Gradient Boosting were the winning models.

Compared to Alper Karaca, Sripad's analysis was short and sweet, consisting of a single cell. His approached differed in that a neural network was used to classify the data. In doing so he returned a 99.13% accuracy for identifying malware. An impressive classification accuracy rate.

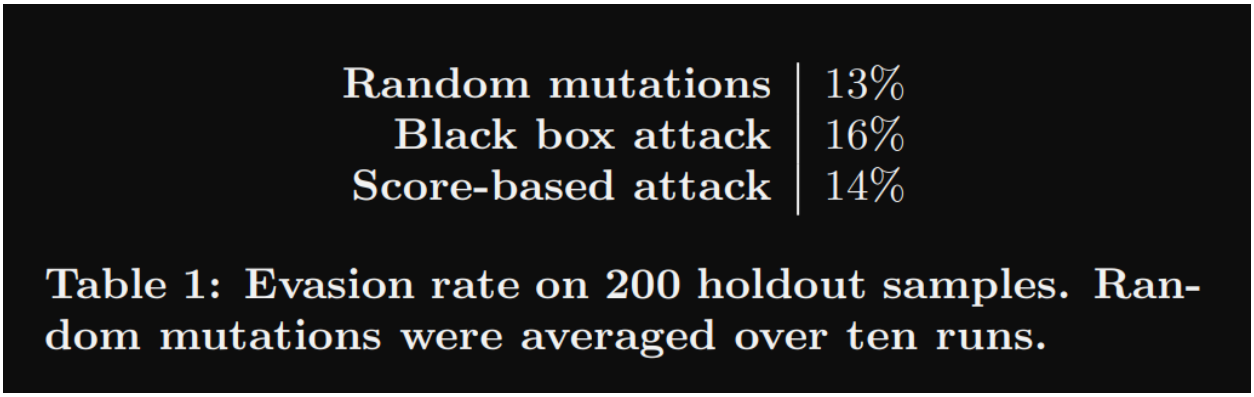
```
Epoch 30/30
2198/2198 [==============================] - 8s 4ms/step - loss: 0.0452 - accuracy: 0.9913
```

Fig 13.

Related Work

Citation: Evading Machine Learning Malware Detection

This paper examines the methods used to evade machine learning detection, with gradient based and other similar methods used. Alongside this, the paper also applies reinforcement learning to their algorithms in hopes of training a model that learns about malware.



Random mutations	13%
Black box attack	16%
Score-based attack	14%

Table 1: Evasion rate on 200 holdout samples. Random mutations were averaged over ten runs.

Fig 14.

The results are given above in table 1. The authors noted that it was surprising the black box attack had the greatest evasion when expectation warranted the score-based attack.

Citation: Performance Evaluation of Machine Learning Algorithms for Detection and Prevention of Malware Attacks

The main focus of the paper is finding an efficient low-cost classification algorithm that is able to detect and stop malware from infecting vulnerable computers. A plethora of classification algorithms were tested to pinpoint one with minimal weakness and greatest accuracy. Malware attack often exploit slow of programs or gaps in security not yet recognized so the algorithm must also be dynamic and possess a small-time complexity.

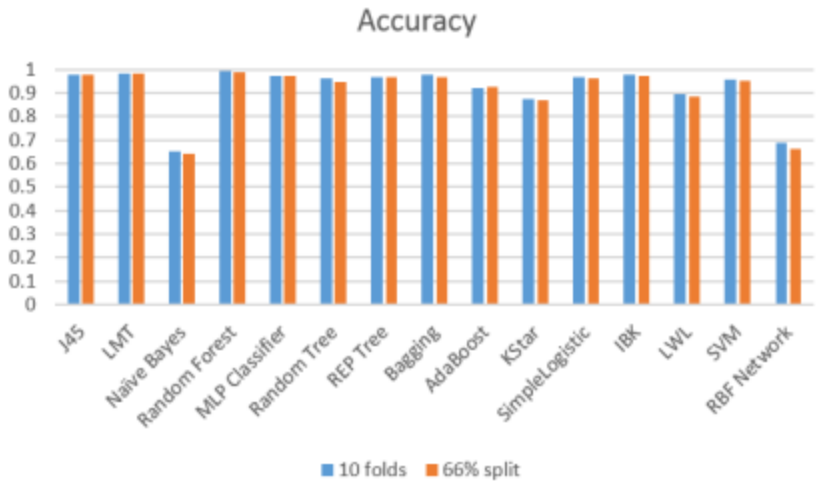


Fig 15.

After testing the 15 different algorithms, it was found that the Random Forest, LMT and J45 generated the highest accuracy with both the 10 folds and 66% split.

Citation: Static Malware Analysis Using Machine Learning Methods

Commonly found features found in malware programs can be utilized to train machine learning models and exploit these recurring traits. After training is complete, the new model can be optimized to find malware programs. When compared to preexisting anti-virus software that do not rely on machine learning, the new machine learning algorithms proposed easily outpaced the status quo.

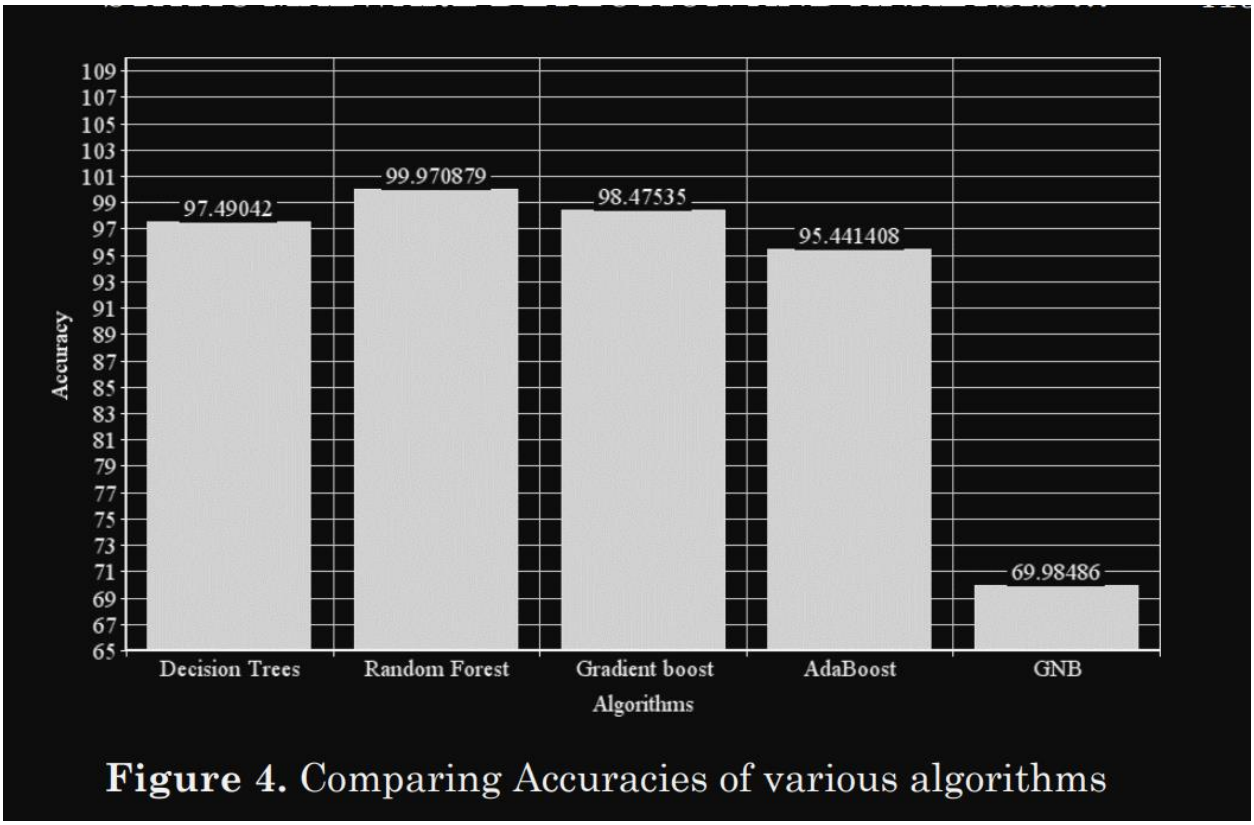


Figure 4. Comparing Accuracies of various algorithms

Fig 16.

Like the other papers, the authors used state of the art and common machine learning algorithms to find malware. The results are very promising and perform at a higher accuracy rate than standard software anti-virus.

Citation: Classification of Malware Attacks Using Machine Learning In Decision Tree

Decision trees are a simple way of classifying objects using a series of steps to determine where an element should be placed. Some of the decisions used by a node in the tree are spam filters, firewalls and IDS/IPS configurations.

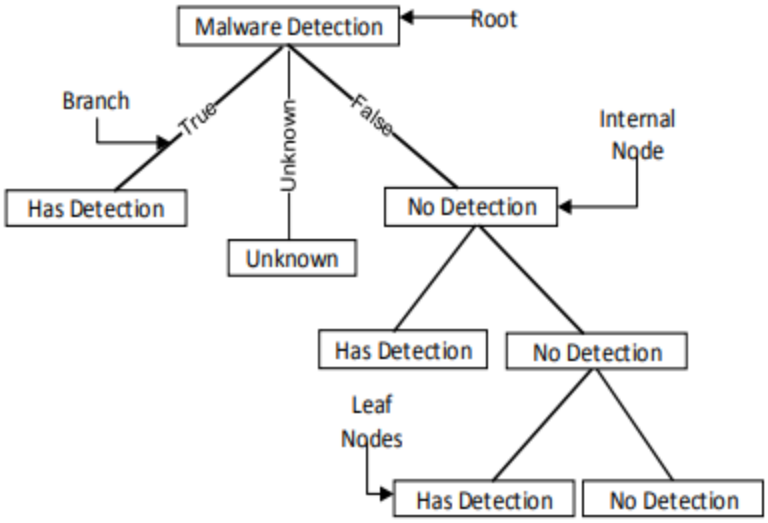


FIGURE 1: Decision Tree.

Fig 17.

As seen in figure 1 above, the decision tree is illustrated to give a visual understanding of how it works. This process was used to detect a variety of different malware with a success range between 55% to 87% for certain types.

Citation: The Curious Case of Machine Learning In Malware Detection

This paper focuses on reviewing already current machine learning techniques used for malware detection and discuss how unique the challenge of finding malware is using state of the art algorithms. Practical challenges discussed by the authors include the cost of training detectors, malware detector interpretability (amount of positives and false positives that can occur) and adversarial malware (malware that can exploit the weaknesses in a detection system).

Feature Selection / Engineering

Principle Component Analysis

Principle Component Analysis (PCA) relies on the linear combination of variables. It therefore examines the correlation between each variable to find a combination of variables. This can be then used to reduce the dimensionality of the model.

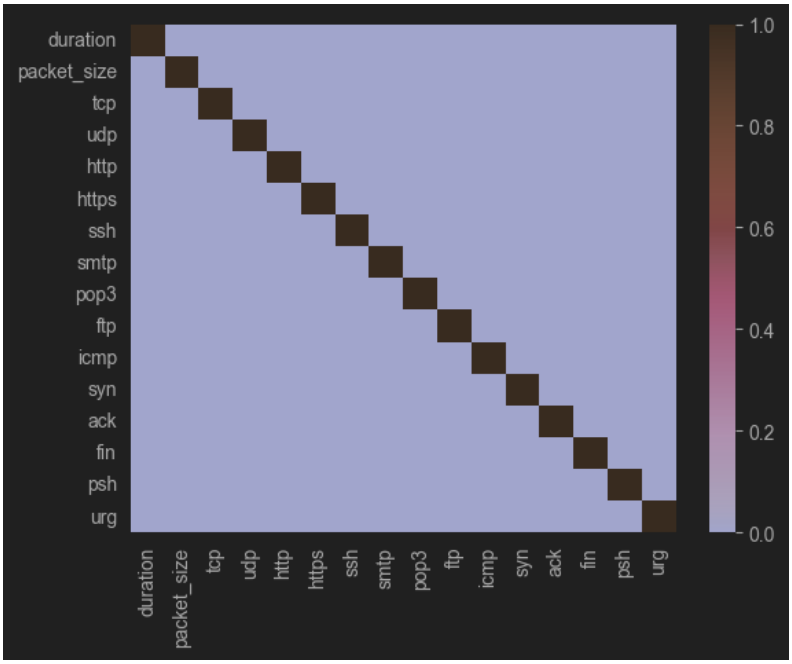


Fig 18.

In Fig 18. All the variables have a correlation of one with themselves but no correlation with any other variables, therefore none can be condensed into a linear combination.

Discretization Transformation

The purpose of discretization is to change a continuous variable into discrete intervals that can be used for later models. Lots of machine learning models perform at a greater rate using discrete data instead of continuous. Another reason for this transformation is to turn numerical data into categorical data.

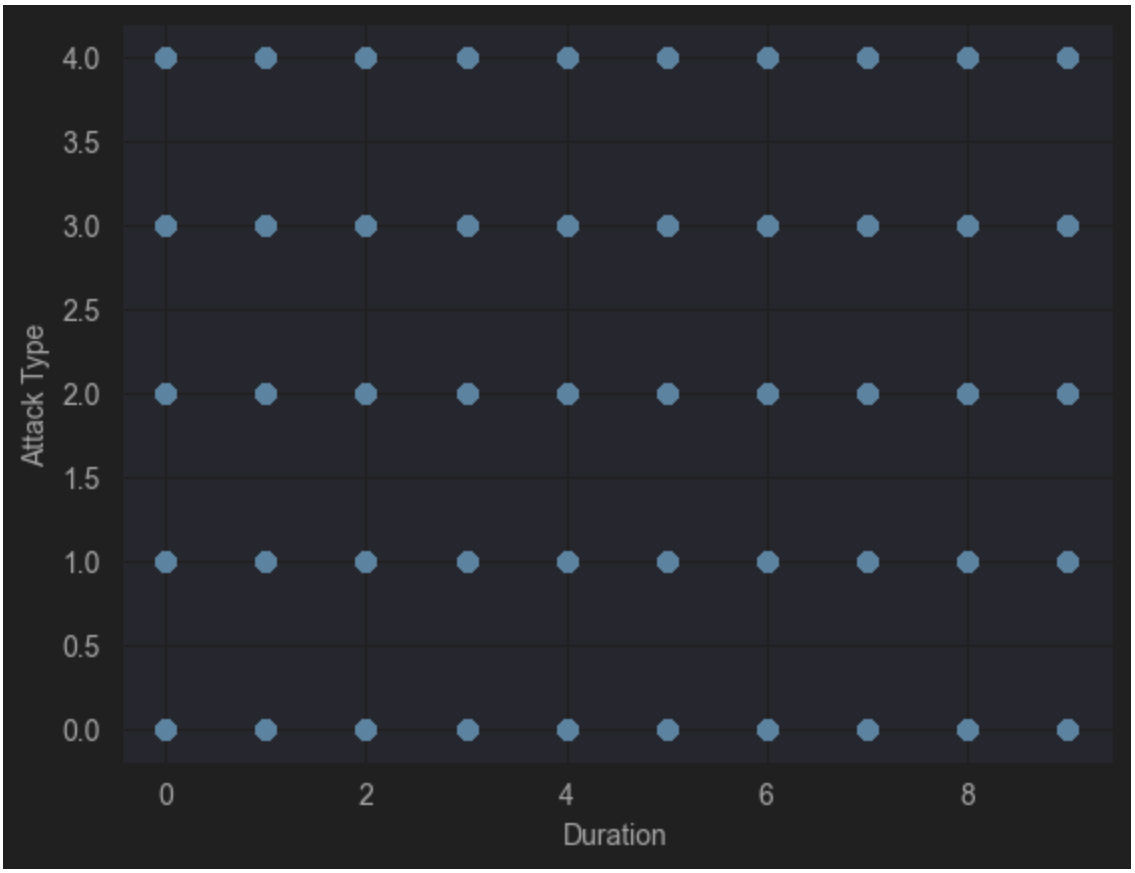


Fig 19.

Above in fig 19 we see that duration has been converted into a discrete variable ranging from 0-9.

Chi-Squared

The general point of the Chi-Squared test is to determine if there is a relationship between the variables. Thus, the null hypothesis states that there is no relationship between the variables while the alternative hypothesis states that there is a relationship between the variables. The p-values are listed below.

p-values:

[0.95448053 0.7843356 0.93425134 0.64901028 0.32483016 0.96806617

0.82236722 0.97182774 0.60942629 0.94140123 0.37798311 0.4216873

0.9691745 0.6731419 0.65003479 0.86276983 0.71343733 0.80952283]

Chi-Squared uses one tail for its test so any p-value below the significance level of 0.05. As seen above, none of the variables have a p-value below the significance level so the null hypothesis is accepted and the alternative rejected.

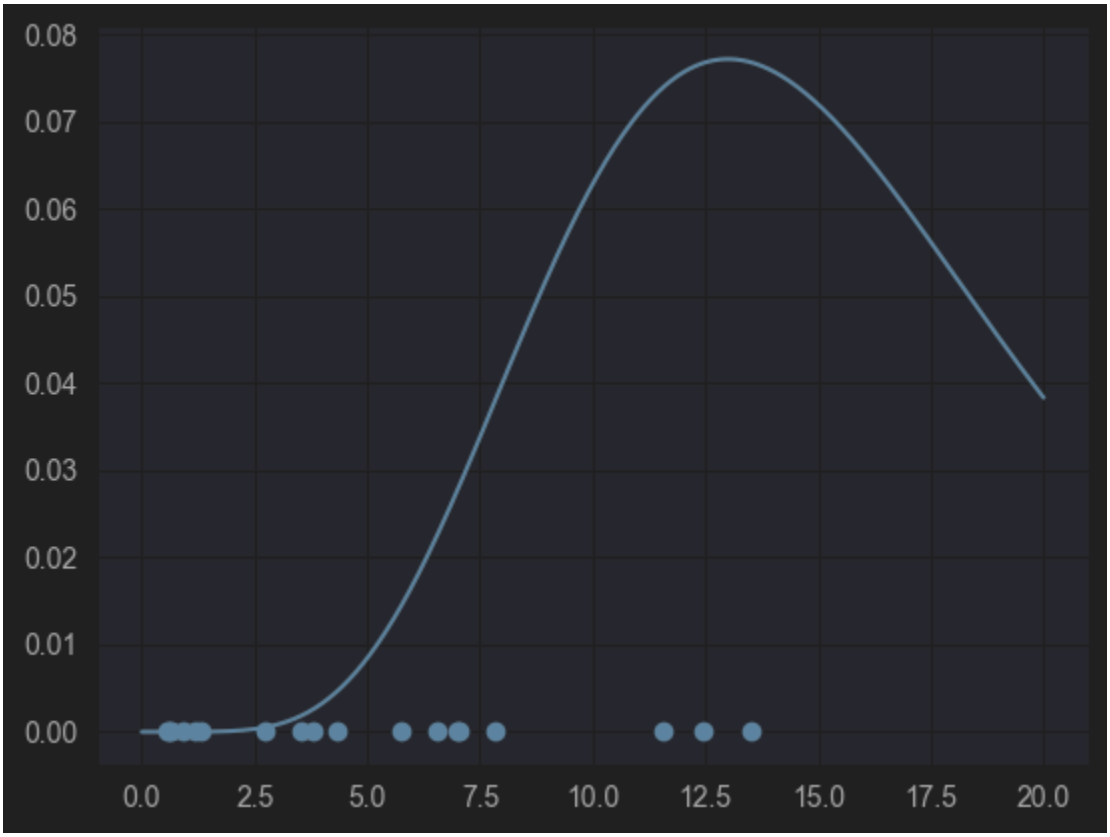


Fig 20.

Comparison

Each feature selection / engineering method serves a specific function and share some similarities. Both PCA and Chi-Squared look for a way to reduce dimensionality by looking for relationships between variables. Linear combination for PCA and mean and variance analysis for Chi-Squared. Discretization Transformation takes a different approach by reducing the range for a continuous variable into a discrete one. This improves the performance of certain machine

learning algorithms such as logistic regression and random forests. As shown and explained above, each transformation has its own purpose, benefits and drawbacks.

Milestone 3

Milestone 3 will focus on the analysis of the original dataset to answer the research question and goal proposed in milestone 1. As a forewarning, the results will be rather underwhelming since the data is uniformly distributed and therefore classes are determined by odds (in this case 1/5). After observing the accuracy rates of the models, the reader will see this is entirely correct.

Independent Variables: ['duration', 'packet_size', 'tcp', 'udp', 'http', 'https', 'ssh', 'smtp', 'pop3', 'ftp', 'icmp', 'syn', 'ack', 'fin', 'psh', 'urg', 'ece', 'cwr']

This is a mixture of continuous and binary data.

Dependent Variable: [‘attack_type’]

Includes 5 classes: DDOS, DOS, Mirai, Recon, Phishing.

Normalized Independent Variables:

	duration	packet_size	tcp	udp	http	https	ssh	smtp	pop3	ftp	icmp	syn	ack	fin	psh	urg	ece	cwr
0	0.928220	0.026678	0	1	0	1	0	1	0	1	0	1	0	0	1	0	0	0
1	0.838779	0.124762	0	1	1	1	1	0	1	0	0	0	1	0	1	0	1	0
2	0.116384	0.529530	1	1	0	1	1	1	0	1	1	0	1	0	0	0	0	1
3	0.055137	0.887629	1	0	0	0	1	1	1	0	1	1	1	1	1	0	0	1
4	0.013467	0.841904	1	1	1	0	1	0	1	0	0	1	0	1	0	0	1	0

Training and Testing Split

A common first step when conducting machine learning analysis is to split the data into a training set and a testing set. This allows us to train our models before using the testing data to check accuracy, ROC-AUC and other metrics. We will be using the standard of separating the data into a 70/30 ratio (training/testing).

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, accuracy_score, recall_score, classification_report
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42)
✓ [23] 131ms
```

Fig 21.

Classical Models

Naïve Bayes (GaussianNB)

The naïve bayes is a supervised learning algorithm that focuses on classification by using a probabilistic approach that relies on the gaussian distribution (normal distribution). The runtime to fit the model was 216 milliseconds. Naïve Bayes has a big O complexity of O(nd), n being the number of training examples and d being the dimensions of the data.

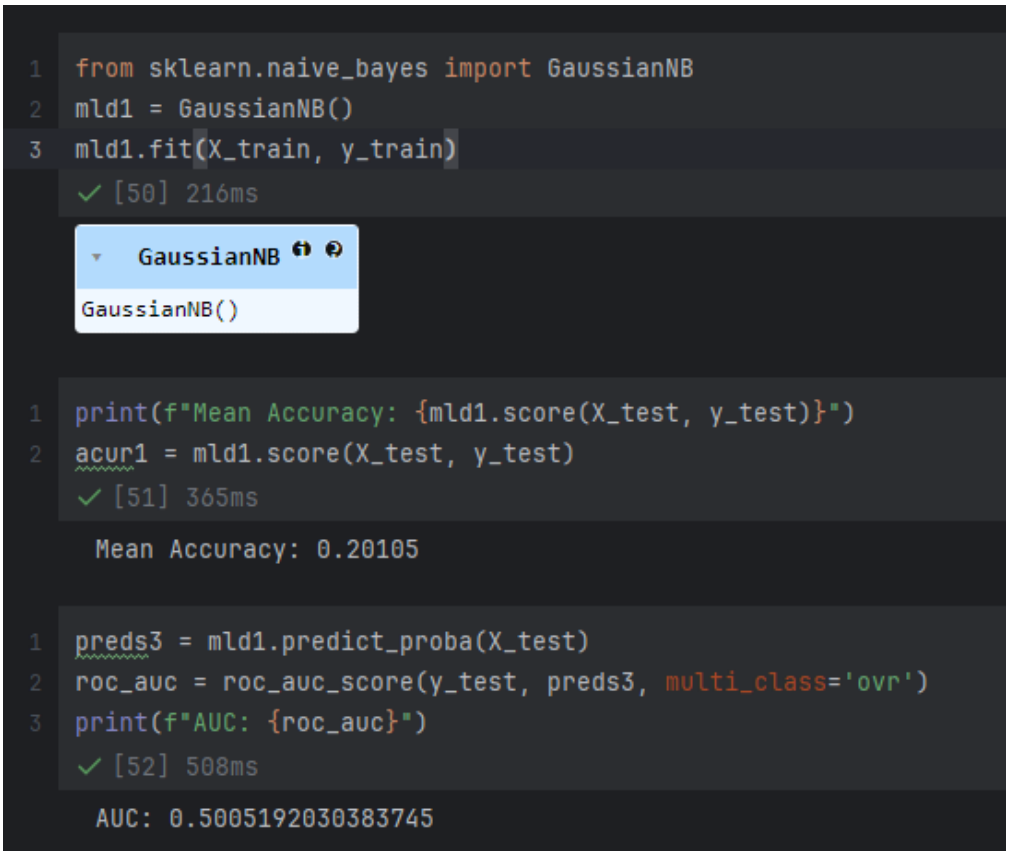


Fig 22.

The score (mean accuracy) when evaluating the X and Y testing data resulted in a .201 or 1/5 chance of being correct. This is expected as the data is uniformed and there is no obvious pattern that can be discerned from the data. Using scikit-learn's built in ROC function we can figure out the area under the curve. This is done by changing the threshold of the function and observing the change in the true positive rate and false positive rate.

```
predClass3 = mld1.predict(X_test)
print(classification_report(y_test, predClass3))
✓ [53] 395ms
```

	precision	recall	f1-score	support
0	0.20	0.36	0.26	48303
1	0.20	0.31	0.25	48210
2	0.20	0.01	0.02	47759
3	0.20	0.22	0.21	47772
4	0.20	0.10	0.13	47956
accuracy			0.20	240000
macro avg	0.20	0.20	0.17	240000
weighted avg	0.20	0.20	0.17	240000

Fig 23.

A summary of other stats was included to flesh out the model's performance on the testing data.

Decision Tree Classifier

Decision Tree Classifier is another supervised machine learning algorithm that utilizes the tree data type to classify which class the data belongs to. There are pros and cons in using a decision tree for classification. pro: the computational complexity is logarithmic, little data preparation only needing normalization, and can handle numerical and categorical data. Cons: tends to overfit the data, piecewise not smooth or continuous, and problems optimizing.

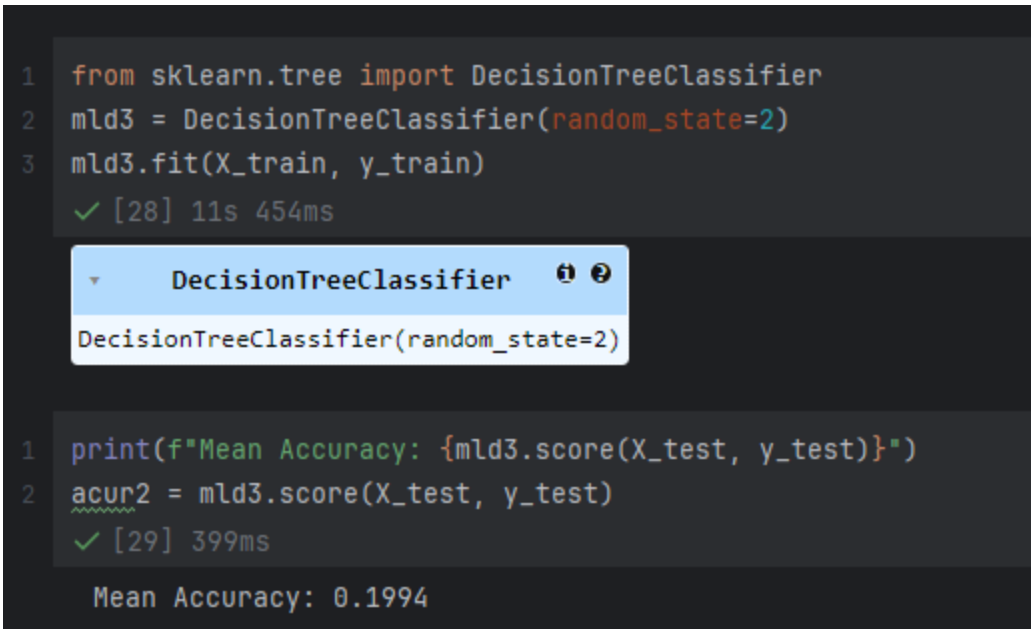


Fig 24.

As with the previous model, the accuracy is only 19.94% after training and then scoring the model.

```
preds2 = mld3.predict_proba(X_test)
roc_auc2 = roc_auc_score(y_test, preds2, multi_class='ovr')
print(f"AUC: {roc_auc2}")
✓ [30] 360ms
```

```
AUC: 0.499625158063536
```

```
predClass2 = mld3.predict(X_test)
print(classification_report(y_test, predClass2))
✓ [31] 440ms
```

	precision	recall	f1-score	support
0	0.20	0.20	0.20	48303
1	0.20	0.20	0.20	48210
2	0.20	0.20	0.20	47759
3	0.20	0.20	0.20	47772
4	0.20	0.20	0.20	47956
accuracy			0.20	240000
macro avg	0.20	0.20	0.20	240000
weighted avg	0.20	0.20	0.20	240000

Fig 25.

The model is guessing at which class the data belongs to which explains the 50% AUC. None of the other stats are particularly interesting as it again is just the random chance of getting the class right.

Random Forest Classifier

The Random Forest Classifier is yet another supervised machine learning algorithm similar to a decision tree, except that it relies on multiple trees to then average out a decision on which class the data belongs to. The biggest benefit for this over a decision tree is that it can be highly

accurate while minimizing overfitting. Big O complexity for random forest classifier is O(v * n log(n)) and fitting the model took 21 seconds and 175 milliseconds.

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X_train, y_train)
print(f"Mean Accuracy {clf.score(X_test, y_test)}")
acur3 = clf.score(X_test, y_test)
✓ [32] 21s 175ms
```

```
Mean Accuracy 0.20127083333333334
```

```
preds = clf.predict_proba(X_test)
roc_auc3 = roc_auc_score(y_test, preds, multi_class='ovr')
print(f"AUC: {roc_auc}")
✓ [54] 1s 143ms
```

```
AUC: 0.5005192030383745
```

Fig 26.

Like the last two classical models, the accuracy and AUC are around 20% and 50%.

```
predClass = clf.predict(X_test)
print(classification_report(y_test, predClass))
✓ [60] 940ms
```

	precision	recall	f1-score	support
0	0.20	0.79	0.32	48303
1	0.20	0.02	0.04	48210
2	0.23	0.01	0.02	47759
3	0.20	0.06	0.09	47772
4	0.20	0.13	0.16	47956
accuracy			0.20	240000
macro avg	0.20	0.20	0.12	240000
weighted avg	0.20	0.20	0.12	240000

Fig 27.

Other statistics included share a similar story to accuracy.

Deep Learning Models

MLP Classifier

The MLP classifier works in a similar way to artificial neural networks in that each step are trained iteratively using the loss function to minimize its output. The two methods used for minimization are limited-memory BFGS or stochastic gradient descent.

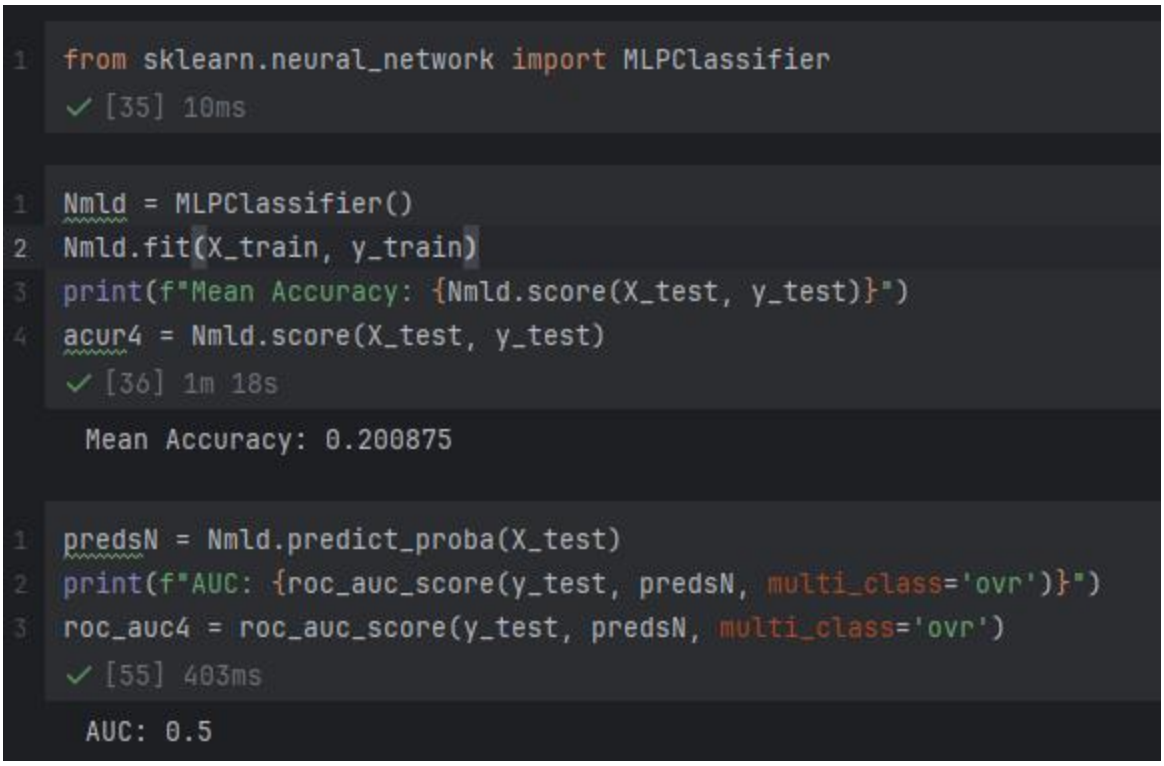


Fig 28.

The accuracy found after fitting the model was 20% while AUC was 50%. A complex model like the MLP classifier still resulted in random chance.

Artificial Neural Network

The artificial neural network was built using the TensorFlow library provided by Google. The model uses multiple hidden layers to analyze the data with different threshold and biases before returning the results in the output layer. After building our layers for the model, we compile and fit the training data to the model. TensorFlow differs from scikit-learn in that it runs multiple epochs to find the global minimum and, in our case, the highest accuracy in classifying attack type correctly.

```
import tensorflow as tf
ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=6,activation="relu"))
ann.add(tf.keras.layers.Dense(units=6,activation="relu"))
ann.add(tf.keras.layers.Dense(units=1,activation="sigmoid"))
✓ [39] 7s 143ms

ann.compile(optimizer="adam",loss="binary_crossentropy",metrics=['accuracy'])
✓ [40] 91ms

ann.fit(X_train,y_train,batch_size=32,epochs = 10)
✓ [41] 1m 49s
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 620us/step - accuracy: 0.1998 - loss: -9149038304928.0000
Epoch 4/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 609us/step - accuracy: 0.1994 - loss: -2471950024704.0000
Epoch 5/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 612us/step - accuracy: 0.1999 - loss: -5165972193280.0000
Epoch 6/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 616us/step - accuracy: 0.1995 - loss: -9334145679360.0000
Epoch 7/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 605us/step - accuracy: 0.2009 - loss: -15218610536448.0000
Epoch 8/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 613us/step - accuracy: 0.1991 - loss: -23256195137536.0000
Epoch 9/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 617us/step - accuracy: 0.1989 - loss: -33655036575744.0000
Epoch 10/10
17500/17500 ━━━━━━━━━━━━━━━━━━━━ 11s 604us/step - accuracy: 0.1991 - loss: -46489850085376.0000
<keras.src.callbacks.history.History at 0x28ce7fd0890>
```

Fig 29.

Above is the code used to build the model as explained in the previous paragraph. When fitting the model, we can see each epoch being ran and time, accuracy and loss that occurs.

Fig 30.

Even when using an advance technique such as artificial neural network, the accuracy still mimics the other models with an accuracy rate of 20%.

Overall Accuracy of All Models

The accuracy of all the models was around the 20% mark. This is to be expected as the data is uniform with no set patterns. If the model were able to predict above the 20% mark (5 classes so 1/5) then there would be something seriously wrong with the model itself. Area under the curve yielded the same information as accuracy with it having a 50% true positive/false positive rate. With that, the models are only randomly choosing a correct result without any insurance of being right.

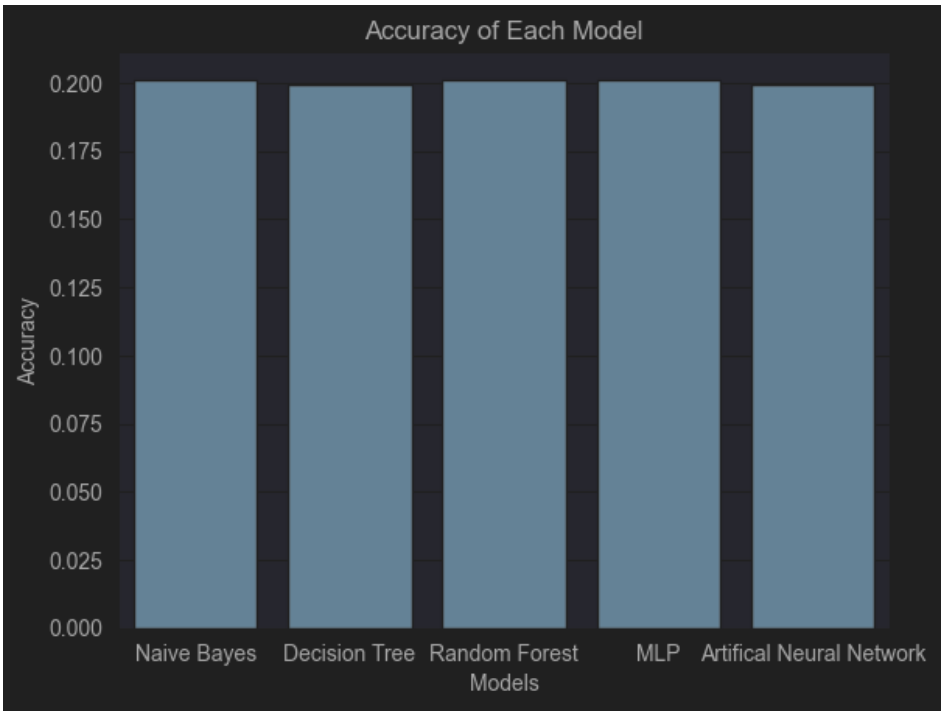


Fig 31.

Runtime of the models

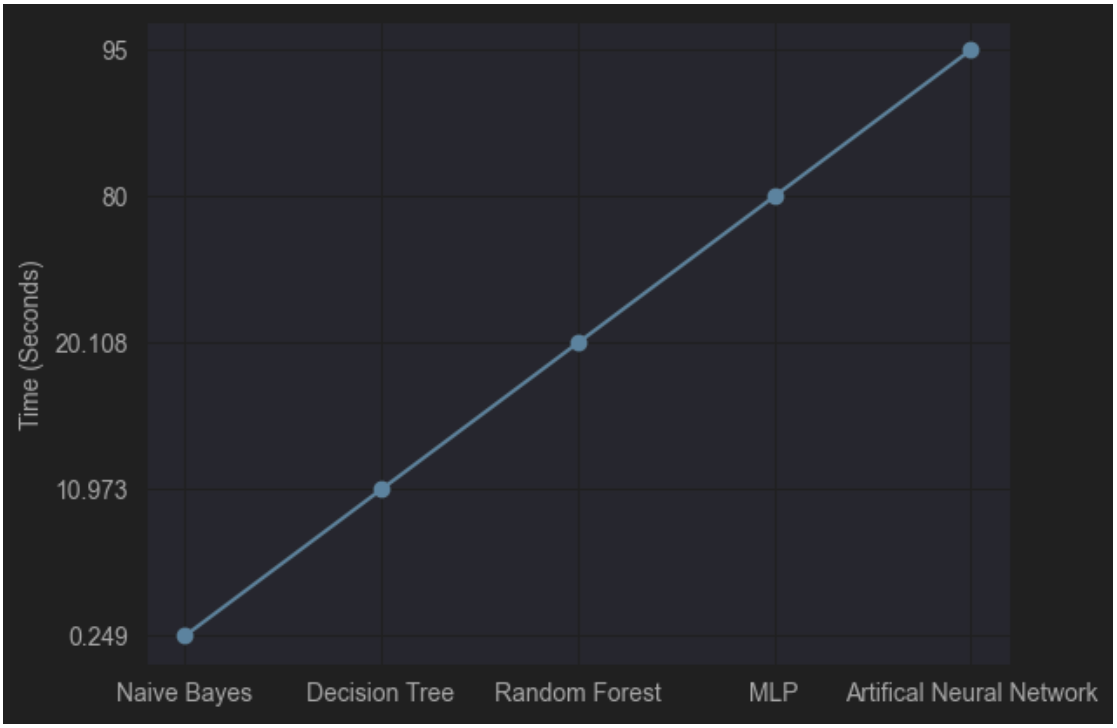


Fig 32.

Based on time to fit the model. As we found that none of the models have a significant difference in accuracy or area under the curve, it would be beneficial to choose the fastest model to fit. This would be Naïve Bayes as it has the fastest experimental time and lowest time complexity.

Comparison with Kaggle Code Sets

It is apparent that all models perform at similar rates when comparing the work of others found on the code set tab of Kaggle. Bosco J Garcia used a logistic regression to classify the data and found that his accuracy rate was 20%, the exact same as our results from the 5 models we tested. The other Kaggle code set by Sadik also ran a bunch of different models similar to our own experiments. These models were CatBoost, AdaBoost, Decision Tree, Random Forest, LightGBM, Extra Tree, Logistic Regression, Ridge and Perceptron. Accuracy and precision for all the models ranged from .19-.2, highlighting that the data set yet again is uniformed and there is no way to accurately predict what attack type will occur. There is no need to compare the other code sets since one is feature engineering and the other two being of a different dataset.

Conclusion

After performing our analysis using different classical and deep learning models, we can fairly say that the results were rather disappointing from a classification standpoint. With an accuracy rate in the 19%-20% range and area under the curve being 50%, none of our models predicted the correct class with any certainty. As mentioned, multiple times throughout the paper, this is because of the uniformed nature of the data. Much like rolling a die, in which the outcome will be of 1/6 chance, our data does a similar thing except it being of 1/5 chance instead. For our research goal and question it is apparent that our outcome from the analysis disproves our hypothesis and therefore there is no independent variables that can predict attack type. Although we disproved our original hypothesis, we can state an alternative hypothesis to counter this. There are no independent variables in the dataset that can determine what type of attack will occur using classification machine learning algorithms without being random chance.