Premihaa_Project_Kitsune.docx

by Premihaa Mohan

Submission date: 17-May-2021 12:44PM (UTC-0700)
Submission ID: 1588240238
File name: 10517-29870-bxgol1_attachment_6356808820210517-29870-l1v2k9.docx (6.48M)
Word count: 3819
Character count: 20789

EE257

MACHINE LEARNING PROJECT: 01

KITSUNE NETWORK ATTACK DATASET

FIRST NAME : PREMIHAA
LAST NAME : RANGARAMANUJAM MOHAN
SJSU ID : 014629316
DATE : 04/09/2021

CONTENTS

A. DATASET DESCRIPTION:

The Kitsune Network Attack Dataset is a collection of nine network attack datasets captured from a either an IP-based commercial surveillance or a network full of IoT devices. Each dataset contains millions of network packets and different cyber-attack within it.

In this Project, The **Mirai Network Dataset** is used. Mirai is a malware that turns networked devices running Linux into remotely controlled bots that can be used as part of a botnet in large-scale network attacks. [Attack name: Mirai, Type: Botnet Malware]

This contains two csv files,

- Mirai_Dataset.csv
- Mirai_labels.csv
- Mirai_pcap.pcap

Mirai_Dataset.csv has 764136 rows entries, 1 to 764136 and 115 columns entries, 1.0 to_0.0.54 . That is [764136 x 115] matrix of 115 sized feature vectors, each describing the packet and the context of that packet's channel with no missing values. The datatype of the dataset is float64(115) and uses 676.3 MB of memory.

Mirai_labels.csv has 764136 x 1 vector of 0-1 values which indicate whether each packet is malicious or benign that is 1 or 0 respectively. There are no missing values in the datasets, out of total packets, 642516 packets are indicated as malicious. The datatype of the dataset is int64 and uses 5.8 MB of memory.

Mirai_pcap.pcap is a raw pcap capture of the original packets. The packets have been truncated to 200 bytes for privacy reasons.

Attributes	115
Input Variables	1.0 to 0.0.54
Output Variables	Benign (0) or Malicious (1)
Missing values	Null
Data types	Float64[Mirai_Dataset.csv] Int64[Mirai_labels.csv]

Next to the object's location, head () returns the first n line. It's useful for fast checking whether the object contains the right kind of data. The top n (default value 5) rows of a data frame or sequence are returned using this form. The following is the top five rows of Mirai dataset which is obtained using head () command.

```
# Changing the attribute numbers to names.
df.columns=["X"+str(i) for i in range(1, 116)]
df.head()
```

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	...	X106	X107	X108	X109	X110	X111	X112	X11
0																			
1	1.999983	60.0	0.000000e+00	1.999990	60.0	9.094947e-13	1.999997	60.0	4.547474e-13	2.000000	...	0.0	0.0	0.0	1.0	60.0	0.0	60.0	0.0
2	1.000000	86.0	0.000000e+00	1.000000	86.0	0.000000e+00	1.000000	86.0	0.000000e+00	1.000000	...	0.0	0.0	0.0	1.0	86.0	0.0	86.0	0.0
3	1.999272	86.0	9.094947e-13	1.999563	86.0	0.000000e+00	1.999854	86.0	9.094947e-13	1.999985	...	0.0	0.0	0.0	1.0	86.0	0.0	86.0	0.0
4	1.000000	60.0	0.000000e+00	1.000000	60.0	0.000000e+00	1.000000	60.0	0.000000e+00	1.000000	...	0.0	0.0	0.0	1.0	60.0	0.0	60.0	0.0
5	1.000000	74.0	0.000000e+00	1.000000	74.0	0.000000e+00	1.000000	74.0	0.000000e+00	1.000000	...	0.0	0.0	0.0	1.0	74.0	0.0	74.0	0.0

5 rows × 115 columns

Fig 1: Top five rows of the Mirai dataset

The Mirai dataset is then merged with the label dataset for the machine learning modeling. The first five rows after merging the dataset is shown below.

	label	X1	X2	X3	X4	X5	X6	X7	X8	X9	...	X106	X107	X108	X109	X110	X111	X112	X113	X
1	0	1.999983	60.0	0.000000e+00	1.999990	60.0	9.094947e-13	1.999997	60.0	4.547474e-13	...	0.0	0.0	0.0	1.0	60.0	0.0	60.0	0.0	
2	0	1.000000	86.0	0.000000e+00	1.000000	86.0	0.000000e+00	1.000000	86.0	0.000000e+00	...	0.0	0.0	0.0	1.0	86.0	0.0	86.0	0.0	
3	0	1.999272	86.0	9.094947e-13	1.999563	86.0	0.000000e+00	1.999854	86.0	9.094947e-13	...	0.0	0.0	0.0	1.0	86.0	0.0	86.0	0.0	
4	0	1.000000	60.0	0.000000e+00	1.000000	60.0	0.000000e+00	1.000000	60.0	0.000000e+00	...	0.0	0.0	0.0	1.0	60.0	0.0	60.0	0.0	
5	0	1.000000	74.0	0.000000e+00	1.000000	74.0	0.000000e+00	1.000000	74.0	0.000000e+00	...	0.0	0.0	0.0	1.0	74.0	0.0	74.0	0.0	

5 rows × 116 columns

Fig 2: Top five rows of the dataset after merging

Pandas dataframe.info() function is used to get a concise summary of the dataframe. It comes really handy when doing exploratory analysis of the data. To get a quick overview of the dataset we use the dataframe.info () function. The summary of Mirai dataset is shown below that displays the number of rows and columns, data types, and memory usage.

```
# Dataset Information
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 764135 entries, 1 to 764135
Columns: 116 entries, label to 0.0.54
dtypes: float64(115), int64(1)
memory usage: 682.1 MB
```

Fig 3: Summary of dataset

The statistical data like percentage, mean, and std of the numerical values of the Series or DataFrame is calculated using describe() method. It analyzes both numeric and object series and also the DataFrame column sets of mixed data types.

```
# Check the Data stats:
data.describe()
```

	label	X1	X2	X3	X4	X5	X6	X7	X8	X
count	764135.000000	764135.000000	764135.000000	764135.000000	764135.000000	764135.000000	764135.000000	764135.000000	764135.000000	764135.0000(
mean	0.840841	62.350842	66.254961	70.099202	89.739233	66.271657	94.559846	240.176311	66.290615	143.84885
std	0.365825	44.615792	17.588507	1131.777296	59.838413	16.923828	1343.726503	141.576086	15.456473	1625.7126(
min	0.000000	1.000000	42.000002	0.000000	1.000000	42.002371	0.000000	1.000000	44.291508	0.0000(
25%	1.000000	7.951423	60.024847	0.236594	10.293483	60.051336	0.592573	22.328584	60.097114	1.36397
50%	1.000000	68.425378	60.078346	0.783347	100.562408	60.110449	1.338655	298.393033	60.135587	1.99952
75%	1.000000	103.520470	60.745052	2.929833	142.195277	60.586338	3.576456	344.376783	61.129885	3.5313(
max	1.000000	144.071660	624.117403	442869.250128	195.788049	590.000000	400508.594984	425.179960	590.000000	265238.69769

8 rows × 116 columns

Fig 4: Statistical data

Using value_counts() method, the number of malignant and benign packets can be identified. In Mirai dataset there are total 764135 labels, in that 642516 are malignant and 121619 are benign packets.

B. DATASET VISUALIZATION:

Every day, trillions of rows of data are created, and visualization is becoming a more important tool for making sense of it. By curating data into an easier-to-understand format and highlighting patterns and outliers, data visualization helps to tell stories. A successful visualization tells a story by eliminating the clutter from data and emphasizing the most important facts.

Common type of data visualization are Charts, Tables, Graphs, Maps. More specifically Scatter plot, Heatmap, Box Plot, Histogram, Matrix are used for data visualization. In this project Heat Map, Scatter Plot, Box Plot, Histogram, Matrix are used to visualization for Mirai dataset.

B.(i): DATA VISUALIZATION USING HEAT MAP:

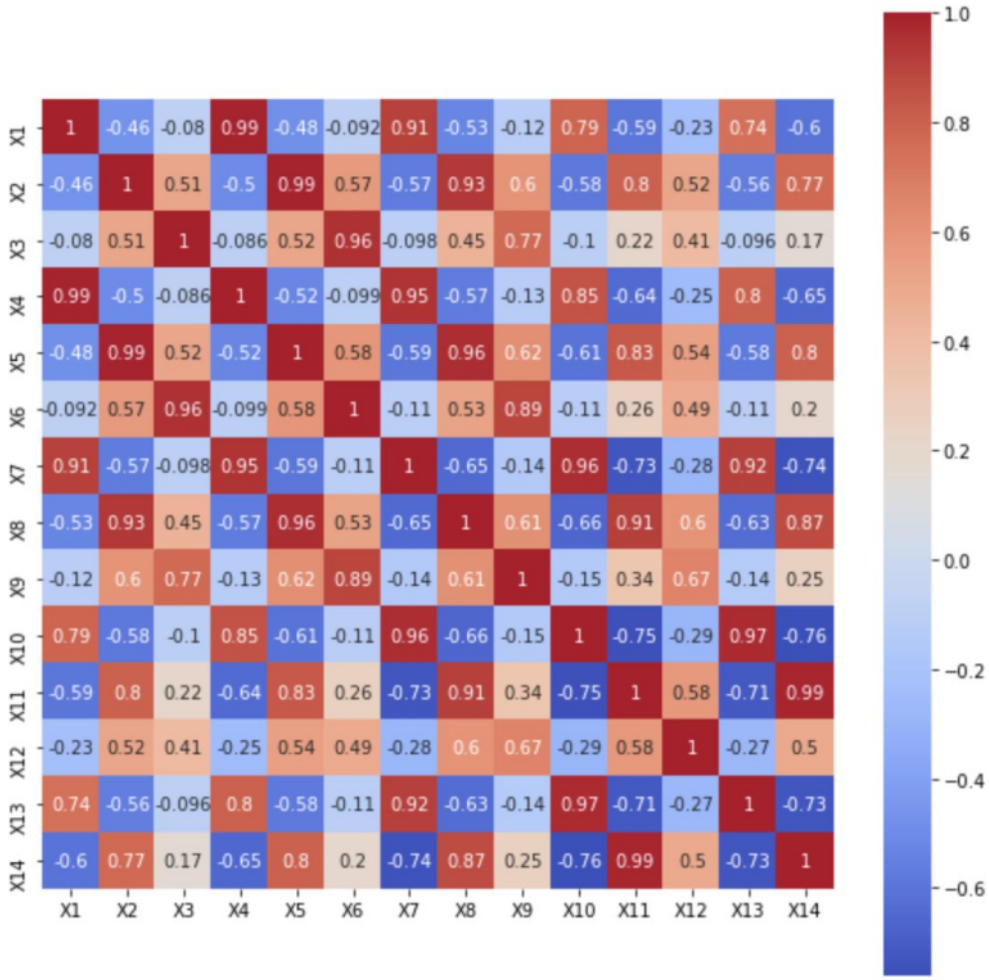

Fig 5: Heat Map

Heat Map shows the correlation between the variables on each axis. Correlation ranges from -1 to +1. Values closer to zero means there is no linear trend between the two variables. The close to 1 the correlation is the more positively correlated they are; that is as one increases so does the other and the closer to 1 the stronger this relationship is. A correlation closer to -1 is similar, but instead of both increasing one variable will decrease as the other increases. The diagonals are all 1/dark red because those squares are correlating each variable to itself (so it's a perfect correlation). For the rest the larger the number and darker the color the higher the correlation between the two variables. The plot is also symmetrical about the diagonal since the same two variables are being paired together in those squares.

B.(ii): DATA VISUALIZATION USING BOX PLOT:

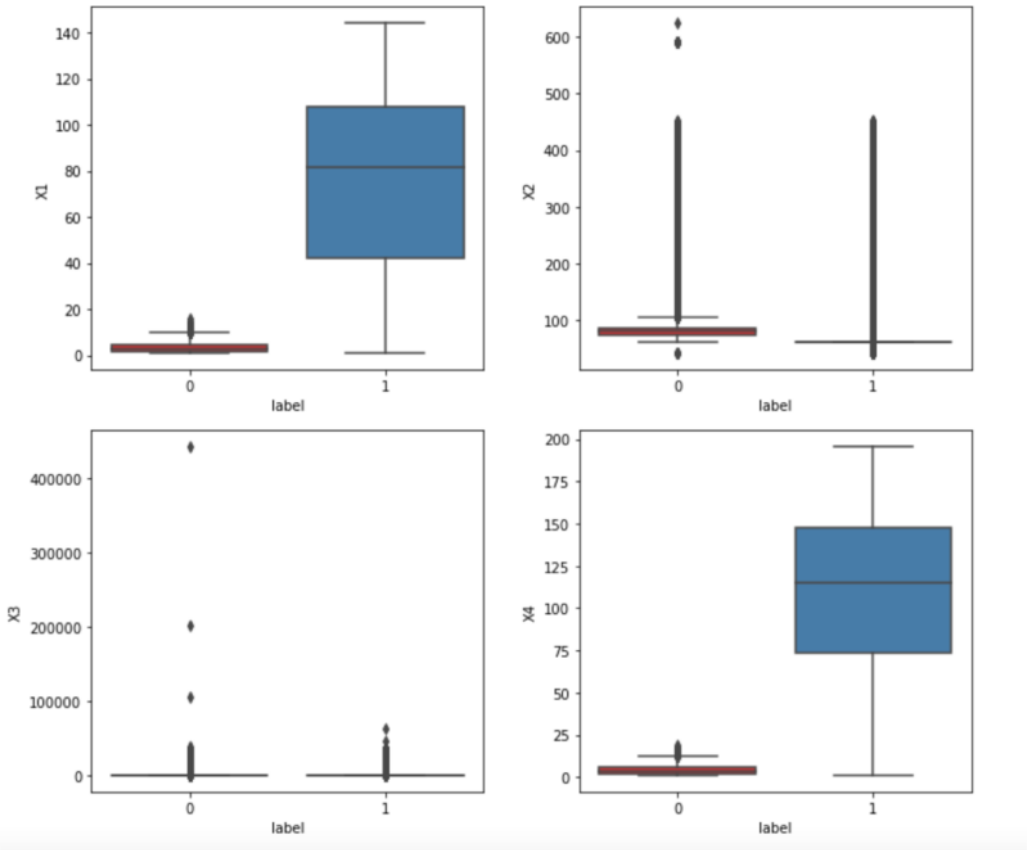

Fig 6: Box Plot

Box plots visually show the distribution of numerical data and skewness through displaying the data quartiles and averages. Box plots show the five-number summary of a set of data: including the minimum score, first (lower) quartile, median, third (upper) quartile, and maximum score. The Fig 6 shows the box plot of four variables X1, X2, X3, X4 with corresponding labels. From the figure we can say that there are many outliers in the dataset.

B.(iii) : DATA VISUALIZATION USING SCATTER MATRIX:

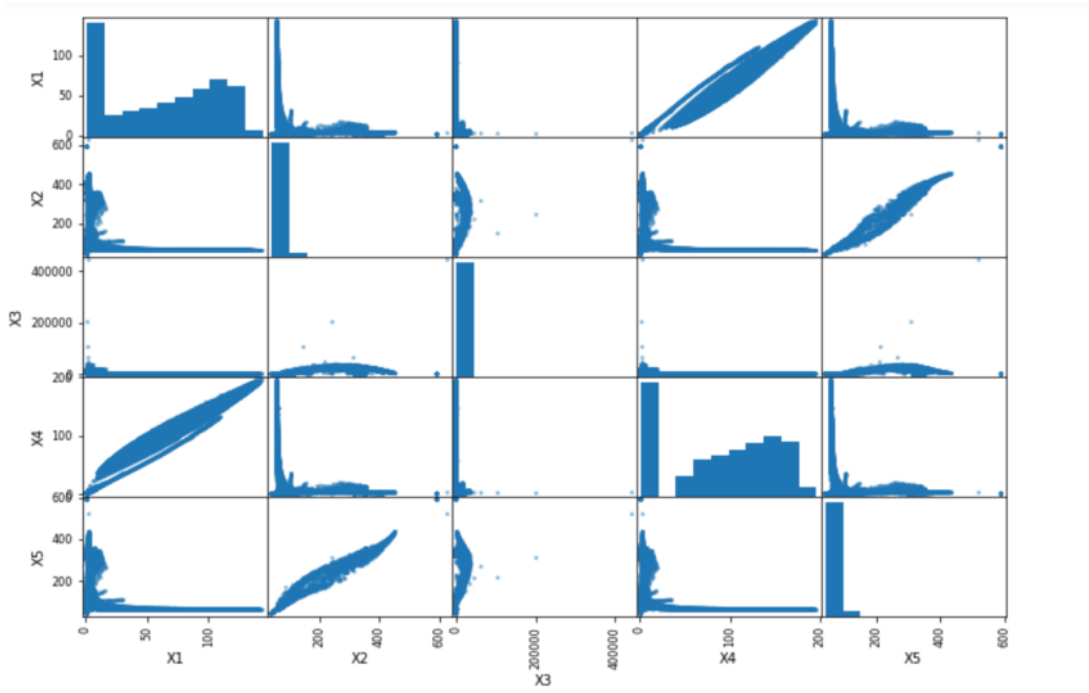

Fig 7: Scatter Matrix

Scatterplot matrices are a great way to roughly determine if you have a linear correlation between multiple variables. The Figure 7 shows the scatter matrix of first five attributes. By looking at the figure, we can say that there are X1 and X4 have linear relationship and also X2 and X5 have somewhat linear relationship.

B.(iv) : DATA VISUALIZATION USING HISTOGRAM:

One of the features that a histogram can show you is the shape of the statistical data. Most of the plot has one tall bar which means all the data may be exactly the same.

```
# Histogram for all 115 attributes:
data.hist(bins=5, figsize=(40,30))
plt.show()
```

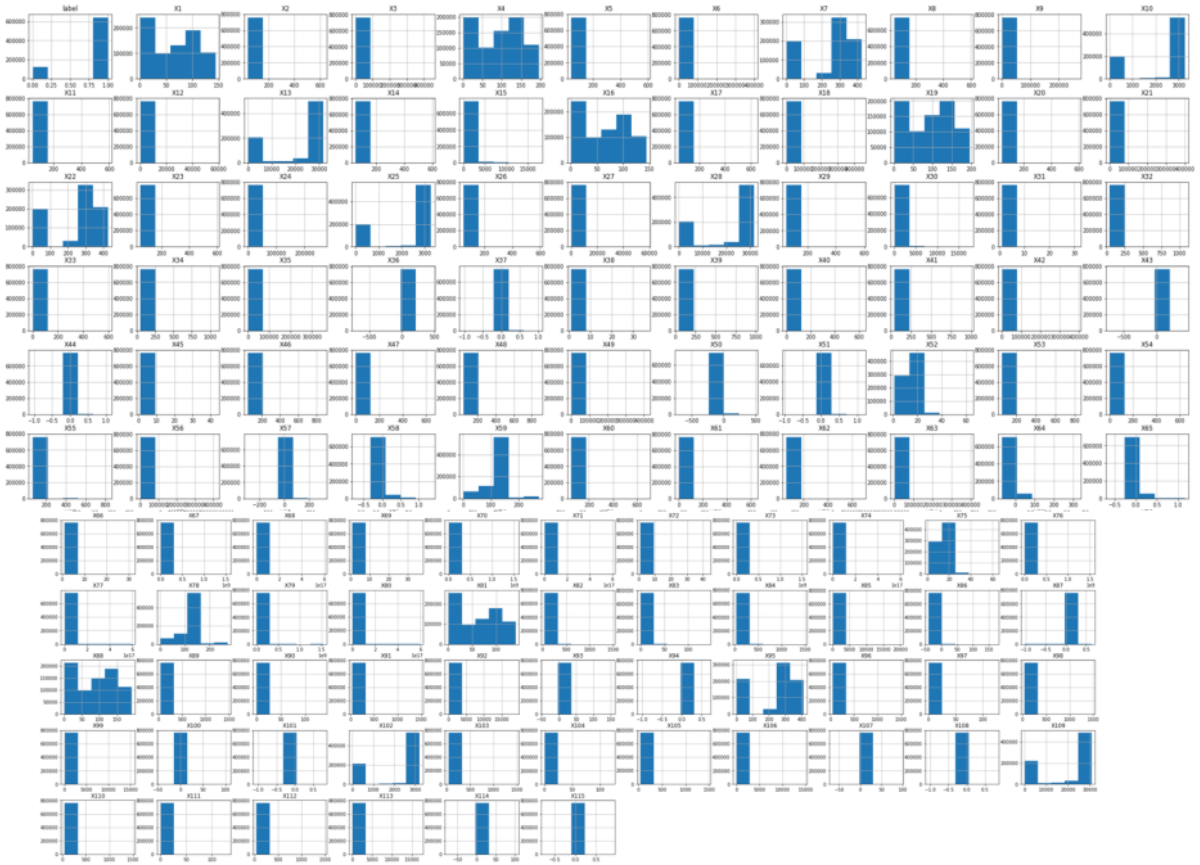

Fig 8: Histogram

B.(v): DATA VISUALIZATION USING SEABORN STRIPPPLOT:

A strip plot is simply a plot of the sorted response values along one axis. The strip plot is an alternative to a histogram or a density plot. It is typically used for small data sets (histograms and density plots are typically preferred for larger data sets).

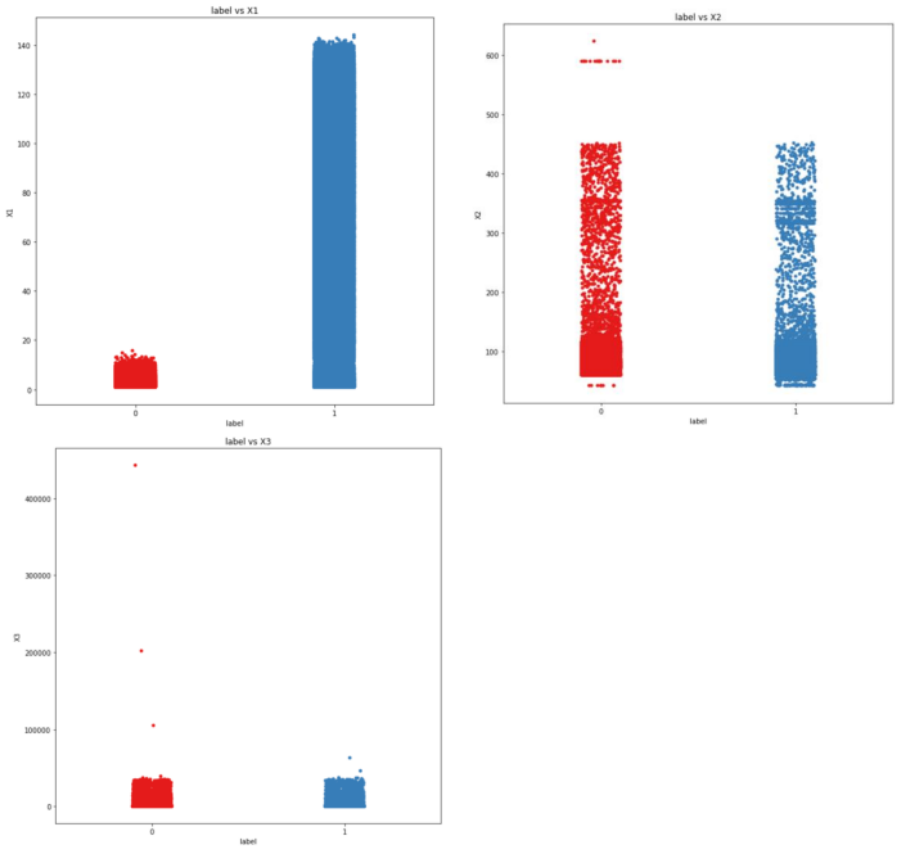

Fig 9: Strip Plot

C. DATASET CLEANING:

Outlier is one that appears to deviate markedly from other members of the sample in which it occurs. From the above plot, one can say that there are many outliers in the dataset. And these outliers are removed after analyzing Z score. The threshold is set to 2 because, lower threshold value removes only few Outliers. Higher the value, removes most of the outliers. In this case, Greater than 2 threshold value removes one whole class of dataset. When the threshold value is set to 3, it removes most of the outliers as well as all the benign packets. Hence the optimal threshold value for Mirai dataset is 2.

After removing most of the outliers, the dataset has 627300 rows.

```
# Handling Outliers:

from scipy import stats
import numpy as np
X = df.iloc[:, 1: ]
z = np.abs(stats.zscore(X))
print(z)
```

```
[[0.35563004 0.06193733 1.46626746 ... 0.05897234 0.00734255 0.00980574]
 [1.12260174 0.06193733 1.48297896 ... 0.05897234 0.00734255 0.00980574]
 [1.12260174 0.06193733 1.46627458 ... 0.05897234 0.00734255 0.00980574]
 ...
 [0.34608295 0.0590653  0.60784421 ... 0.05897234 0.00734255 0.00980574]
 [0.34652563 0.05919737 0.59438706 ... 0.05897234 0.00734255 0.00980574]
 [2.91085256 0.01852978 1.47743809 ... 0.05897234 0.00734255 0.00980574]]
```

```
print(np.count_nonzero(z > 2))
threshold = 2
print(np.where(z>2))
```

```
1792043
(array([     0,      0,      0, ..., 764135, 764135, 764135]), array([ 50,  57,  65, ..., 103, 108, 110]))
```

```
ds = X[(z < 2).all(axis=1)]
print(ds.shape)
print(ds.index)
```

```
(627300, 114)
Int64Index([  1237,   1238,   1239,   1246,   1247,   1248,   1258,   1259,
              1260,   1261,
             ...
             764125, 764127, 764128, 764129, 764130, 764131, 764132, 764133,
             764134, 764135],
            dtype='int64', name='0', length=627300)
```

Fig 10: Removing outliers

D. RESEARCH WORK:

Botnet Detection Using Machine Learning Techniques - Nadia Chaabouni [1] implements a Network Intrusion Detection System (NIDS) based on machine learning techniques for botnet detection. They conducted a survey of numerous papers in their research and used the results to apply various machine learning algorithms to a variety of public datasets for their study. Machine learning techniques are used by Ayush Kumar and Teng Joon Lim [2] to detect IoT malware networks early. They build their own dataset by using testbed experiments and packet capture utilities to analyze traffic patterns for IoT malware. They used gateway-level traffic classification for early detection, including only TCP packets with the SYN flag enabled. Zhipeng Liu [3] suggested an anomaly detection method based on machine learning by examining a new dataset called the IoT Network Intrusion Dataset and conducting experiments with a variety of machine learning algorithms. They used AI speakers and smart cameras to create the dataset. Convolutional Features and Neural Networks are used by Shao-Chien Chen [4] to detect botnets. They extract a number of flow-based features from the packet header and use them to train the neural network. They demonstrated that by using convolutional features, they were able to provide a 94.7 percent accuracy for P2P botnets. Rohan

Doshi [5] uses machine learning 8 algorithms and neural networks to detect botnets using IoT-specific network behaviors. For the study, they generated their own dataset using a network of simulated IoT devices. Data collection, feature extraction, and binary classification for traffic data containing DDoS attacks are all possible with their pipeline. The primary goal of the paper is to identify the IoT botnets at local network level.

E. FEATURE EXTRACTION:

There are many different types of Feature extraction methods. In this project PCA – Principle Component Analysis method is used for feature extraction.

Principle Component Analysis (PCA) is a common feature extraction method in data science. Technically, PCA finds the eigenvectors of a covariance matrix with the highest eigenvalues and then uses those to project the data into a new subspace of equal or less dimensions. Practically, PCA converts a matrix of n features into a new dataset of (hopefully) less than n features. That is, it reduces the number of features by constructing a new, smaller number variables which capture a significant portion of the information found in the original features.

```
from sklearn.decomposition import PCA
pc = PCA(n_components=4)
principalComponents = pc.fit_transform(X)
per_var = np.round(pc.explained_variance_ratio_* 100, decimals=0)

labels = ['PC' + str(x) for x in range(1, len(per_var)+1)]

plt.bar(x=range(1,len(per_var)+1), height=per_var, tick_label=labels,  align="center")
#plt.xticks(range(1, len(principalComponents)+1), principalComponents)
plt.ylabel('Percentage of Explained Variance')
plt.xlabel('Principal Component')
plt.title('Scree Plot')
plt.show()
```

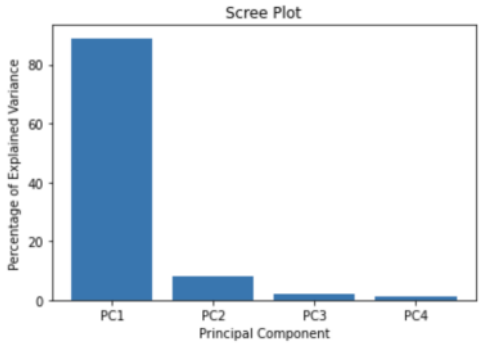

Fig 11. Principle Component Analysis

The Fig 10, Shows the percentage of explained variance vs Principle Component. The Principle Component 1 has the highest variance about 89 percent and Principle component 2 has the second highest variance equal to 8 percent. Rest all Principle component has minimal variance and they are ignored. Hence Principle component 1 and Principle component 2 are used for further data analysis.

	principal component 1	principal component 2
1	19.750406	-0.183191
2	19.717748	-0.172309
3	19.688908	-0.163250
4	19.507050	-0.140303
5	19.480181	-0.133088
...	...	...
627296	-1.207331	-0.042877
627297	-1.216286	-0.041094
627298	-1.234473	-0.040044
627299	-1.252088	-0.038791
627300	-1.265765	-0.037509

627300 rows × 2 columns

Fig 12: Two Principle Components

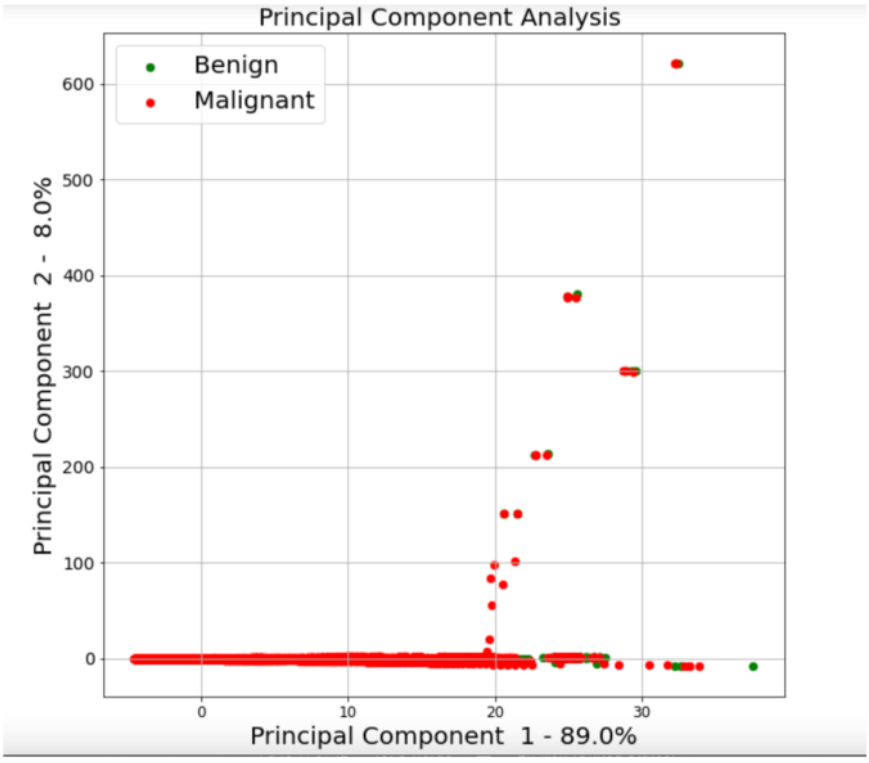

Fig 13: Principal Component Analysis Plot

F. MODEL DEVELOPMENT:

Since the dataset has been classified into two classes, Benign or Malicious Packet, it is a Classification problem. A classification can have real-valued or discrete input variables. A problem with two classes is often called a two-class or binary classification problem. There are many different types of classification methods, in this project we use Logistic regression, KNN Classifier, Decision tree classifier and Support vector machine Classifier.

Since the raw dataset has huge number of data points, it takes long time to execute or sometimes kernel gets restarted/killed. Hence using raw dataset is impossible in this situation. So, the Extracted features from the Principle component analysis method is used for further analysis.

F. (1) LOGISTIC REGRESSION MODEL:

Logistic Regression is widely used for binary classification like (0,1). The binary logistic model is used to estimate the probability of a binary response based on one or more predictor (or independent) variables (features).

```
In [93]: from sklearn.preprocessing import StandardScaler
         # Dividing X, y into train and test data
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state = 0)
         # Standardizing the features
         sc = StandardScaler()
         X_train = sc.fit_transform(X_train)
         X_test = sc.fit_transform(X_test)
```

```
In [95]: classifier = LogisticRegression()
         classifier.fit(X_train, y_train)
         print("LogisticRegression")

         LogisticRegression
```

```
In [106]: # Predicting the Train set results
          y_trainpred = classifier.predict(X_train)
          print(y_trainpred)
```

Fig 14: Logistic Regression

```
# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_train,y_trainpred)*100)


#Test MSE
print ('Test MSE: ', mean_squared_error(y_train, y_trainpred))

#Root_Mean_Squared_Error:
print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_train,y_trainpred)))
```

```
[1 0 1 ... 1 1 1]
Accuracy is 94.24244494545786
Test MSE:  0.05757555054542142
Root_Mean_Squared_Error :  0.23994905822991142
```

```
#Classification report

print("Classification report :")
print(classification_report(y_train,y_trainpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.62      0.78      0.69     36485
           1       0.98      0.96      0.97    402625

    accuracy                           0.94    439110
   macro avg       0.80      0.87      0.83    439110
weighted avg       0.95      0.94      0.95    439110
```

Fig 15: Performance Metrics of Logistic Regression (Train set)

Precision is how certain you are of your true positives. Recall is how certain you are that you are not missing any positives. From the above metrics, we can conclude that, Accuracy score value of 94 percent means identification of 6 of every 100 packets is incorrect, and 94 is correct. Precision value of 62 percent means, 62 of every 100 benign packets are predicted as benign and remaining are predicted as malicious which is incorrect. Similarly, performance metrics of test dataset are shown below in the Figure 16.

A confusion matrix is a matrix representation of the prediction results of any binary testing that is often used to describe the performance of the classification model (or "classifier") on a set of test data for which the true values are known.

```
# Predicting the Test set results
y_testpred = classifier.predict(X_test)
print(y_testpred)

# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_test,y_testpred)*100)

#Test MSE
print ('Test MSE: ', mean_squared_error(y_test, y_testpred))


print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_test,y_testpred)))
```

```
[1 1 1 ... 1 1 0]
Accuracy is 94.19257133747809
Test MSE:  0.05807428662521919
Root_Mean_Squared_Error :  0.24098607143405446
```

```
#Classification report

print("Classification report :")
print(classification_report(y_test,y_testpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.62      0.78      0.69     15788
           1       0.98      0.96      0.97    172402

    accuracy                           0.94    188190
   macro avg       0.80      0.87      0.83    188190
weighted avg       0.95      0.94      0.94    188190
```

```
#confusion matrix of Logistic Regression
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

Predicted \ Actual	0	1
0	12258	7399
1	3530	165003

Fig 16: Performance Metrics of Logistic Regression (Test set)

F. (2) DECISION TREE CLASSIFIER:

A decision tree is a flowchart-like tree structure in which an internal node represents a function (or attribute), a branch represents a decision law, and each leaf node represents the result. The root node is the topmost node in a decision tree. It learns to partition based on the value of an attribute.

```
clf = DecisionTreeClassifier(max_depth=6)
clf.fit(X_train, y_train)
```

```
DecisionTreeClassifier(max_depth=6)
```

```
from sklearn import tree
from graphviz import Source


dot_data = StringIO()

export_graphviz(clf, out_file=dot_data, filled=True, rounded=True, special_characters=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

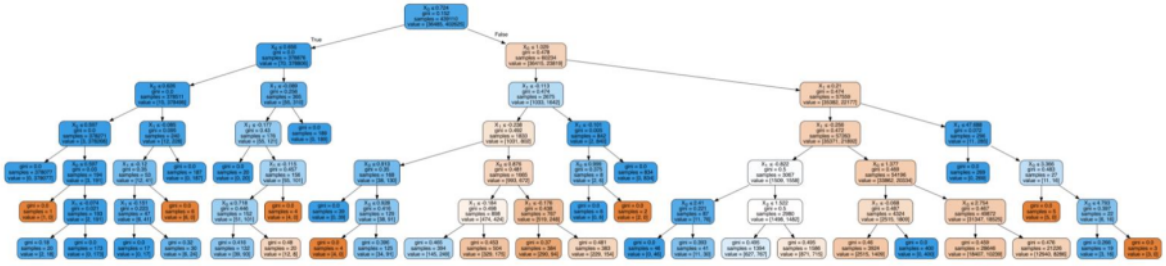

Fig 17: Decision Tree

Pruning reduces the size of decision trees by removing parts of the tree that do not provide power to classify instances. Decision trees are the most susceptible out of all the machine learning algorithms to overfitting and effective pruning can reduce this likelihood.

```
SCORES = []
max_leafs_arr = range(2, 50)
for max_leafs in max_leafs_arr:
    regressionTree = DecisionTreeClassifier(max_leaf_nodes=max_leafs)
    sc = cross_val_score(regressionTree, X, y, cv=10, scoring="neg_mean_squared_error")
    SCORES.append((-sc.mean(), sc.std()))
SCORES = np.array(SCORES)
```

```
plt.xkcd()
plt.figure(figsize=(25, 10))
plt.plot(max_leafs_arr, SCORES[:,0], 'g')
plt.fill_between(max_leafs_arr, SCORES[:,0]+SCORES[:,1], SCORES[:,0]-SCORES[:,1], alpha=0.3, color='y')
plt.xlabel('tree size', fontsize=20, color='c')
plt.ylabel('MSE', fontsize=20, color='c')
plt.title('finding the best tree through cross-validation', fontsize=30, color='m')
best_min_leafs = max_leafs_arr[np.argmin(SCORES[:,0])]
print(f"The best tree has {best_min_leafs} leafs.")
```

```
The best tree has 31 leafs.
```

Fig 18: Pruning

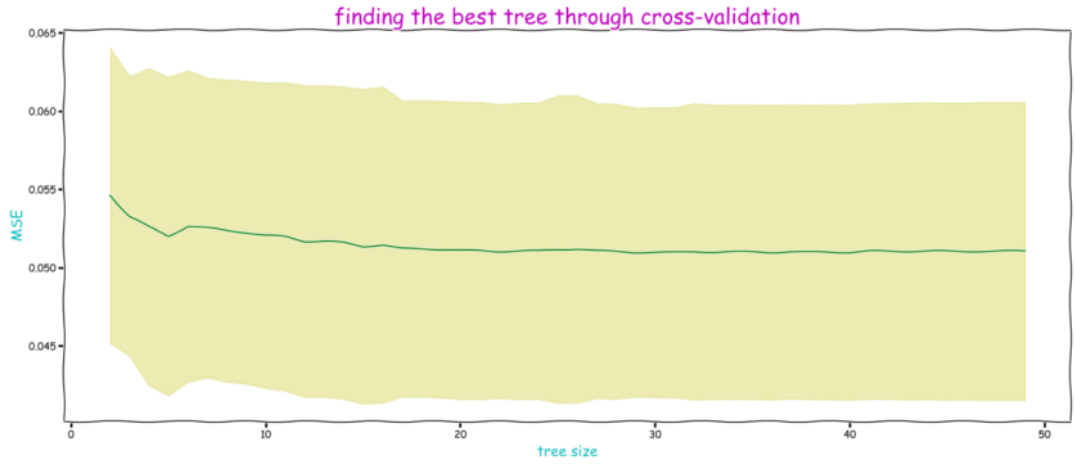

Fig 19: Finding the best tree through CV

From the above plot, we can say that X-axis tree size at 31 or above has the lowest MSE, Mean Square Error. Hence tree size 31 is preferred. Pruning can not only significantly reduce the size but also improve the classification accuracy of unseen objects. It may be the case that the accuracy of the assignment on the train set deteriorates, but the accuracy of the classification properties of the tree increases overall.

```
### ***Train Set*** ####
# Model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_train,y_trainpred)*100)


#Test MSE
print ('Test MSE: ', mean_squared_error(y_train, y_trainpred))

#Root_Mean_Squared_Error:
print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_train,y_trainpred)))
```

```
[1 0 1 ... 1 1 1]
Accuracy is 95.00193573364305
Test MSE:  0.04998064266356949
Root_Mean_Squared_Error :  0.22356350923970014
```

```
#Classification report - Decision tree classifier

print("Classification report :")
print(classification_report(y_train,y_trainpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.63      0.98      0.76     36485
           1       1.00      0.95      0.97    402625

    accuracy                           0.95    439110
   macro avg       0.81      0.96      0.87    439110
weighted avg       0.97      0.95      0.95    439110
```

Fig 20: Performance metrics of Decision tree Classifier (Train set)

F. (3) SUPPORT VECTOR MACHINE MODEL:

SVM is a fascinating algorithm with straightforward principles. The classifier uses the hyperplane with the most margin to distinguish data points. An SVM classifier is also known as a discriminative classifier because of this. SVM determines the best hyperplane for classifying new data points.

As compared to other classifiers such as logistic regression and Decision tree, SVM has a very high accuracy. It is known for its kernel trick to handle nonlinear input spaces. To divide different groups, SVM creates a hyperplane in multidimensional space. SVM iteratively produces the best hyperplane, which is then used to minimize an error. The aim of SVM is to find a maximum marginal hyperplane (MMH) that divides a dataset into classes as evenly as possible. The figure 21 given below shows how to split and train the data in SVM model in python.

```
# Dividing X, y into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 0)
# Standardizing the features
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
```

```
lab_enc = preprocessing.LabelEncoder()
training_scores_encoded = lab_enc.fit_transform(y_train)
print(training_scores_encoded)
print(utils.multiclass.type_of_target(y_train))
print(utils.multiclass.type_of_target(y_train.astype('int')))
print(utils.multiclass.type_of_target(training_scores_encoded))
```

```
[1 1 1 ... 1 1 1]
binary
binary
binary
```

```
C = 1.0
# training a linear SVM classifier
svm_model_linear = SVC(kernel = 'linear', C=C).fit(X_train,training_scores_encoded)
```

Fig 21: SVM Model

```
# Predicting the Train set results
y_trainpred = svm_model_linear.predict(X_train)

print(y_trainpred)


# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_train,y_trainpred)*100)


#Test MSE
print ('Test MSE: ', mean_squared_error(y_train, y_trainpred))

#Root_Mean_Squared_Error:
print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_train,y_trainpred)))
```

```
[0 1 1 ... 1 1 1]
Accuracy is 93.15576923076922
Test MSE:  0.06844230769230769
Root_Mean_Squared_Error :  0.2616148078613053
```

```
#Classification report

print("Classification report :")
print(classification_report(y_train,y_trainpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.75      0.90      0.82     17825
           1       0.98      0.94      0.96     86175

    accuracy                           0.93    104000
   macro avg       0.86      0.92      0.89    104000
weighted avg       0.94      0.93      0.93    104000
```

Fig 22: Performance metrics of SVM Classifier (Train set)

F. (4) KNN MODEL:

KNN is a non-parametric learning algorithm, which means it makes no assumptions about the distribution of the underlying data. KNN stands for the number of closest neighbors. The number of neighbors is the most important consideration. If there are two groups, K is usually an odd number. The algorithm is known as the nearest neighbor algorithm when K=1. This is the most basic scenario. Suppose P1 is the point, for which label needs to predict. First, you find the one closest point to P1 and then the label of the nearest point assigned to P1.

```
# Dividing X, y into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.3, random_state = 42)
# Standardizing the features
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)

knn = KNeighborsClassifier(n_neighbors = 13)
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier(n_neighbors=13)
```

Fig 23: KNN algorithm

```
# Predicting the Train set results
y_trainpred = knn.predict(X_train)

print(y_trainpred)


# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_train,y_trainpred)*100)


#Test MSE
print ('Test MSE: ', mean_squared_error(y_train, y_trainpred))

#Root_Mean_Squared_Error:
print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_train,y_trainpred)))
```

```
[1 1 1 ... 1 1 1]
Accuracy is 95.61954863246112
Test MSE:  0.04380451367538885
Root_Mean_Squared_Error :  0.2092952786743859
```

```
#Classification report

print("Classification report :")
print(classification_report(y_train,y_trainpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.69      0.87      0.77     36520
           1       0.99      0.96      0.98    402590

    accuracy                           0.96    439110
   macro avg       0.84      0.92      0.87    439110
weighted avg       0.96      0.96      0.96    439110
```

Fig 24: Predicting train set

G. FINE TUNE THE MODELS AND FEATURE SET:

A model hyperparameter is a feature of a model that is external to the model and cannot be estimated from data. Before the learning process can begin, the hyperparameter's value must be chosen. The number of hidden layers in Neural Networks, for example, is c in Support Vector Machines and k in k-Nearest Neighbors. A parameter, on the other hand, is an internal characteristic of the model whose value can be calculated using data. For example, linear/logistic regression beta coefficients or support vectors in Support Vector Machines.

In this project **Grid-search CV** is used to find the optimal hyperparameters of a **logistic regression model** which results in the most 'accurate' predictions.

```
X = Df_new1.iloc[:, 0:2].values
y=Df_new1.label.values
X

from sklearn.preprocessing import StandardScaler
# Dividing X, y into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state = 0)
# Standardizing the features
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Setup the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

# Instantiate a logistic regression classifier: logreg
logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

# Fit it to the data
logreg_cv.fit(X_train, y_train)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

```
Tuned Logistic Regression Parameters: {'C': 0.006105402296585327}
Best score is 0.942422172120881
```

Fig 25: Fine Tuning Logistic Regression Model

By fitting the Logistic Regression model with the default parameters, we have a much 'better' model. The accuracy is 94.24% and at the same time, the Precision is a staggering 98%. Now, let's take a look at the confusion matrix again for this model results again:

```
# Predicting the Test set results
y_testpred = logreg_cv.predict(X_test)
print(y_testpred)


#confusion matrix of Logistic Regression
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

```
[1 1 1 ... 1 1 0]
```

Actual	0	1
Predicted		
0	12258	7398
1	3530	165004

Fig 26: Confusion matrix

Looking at the misclassified instances, we can observe that 3530 malignant cases have been classified incorrectly as benign (False negatives). Also, 7398 benign case has been classified as malignant (False positive).

We can tune k, the number of nearest neighbors, in the case of **k-NN**. Our distance metric/similarity function can also be tweaked.

RANDOMIZED SEARCH HYPERPARAMETER:

The **Random Search method of hyperparameter tuning** uses a random, uniform distribution to sample hyperparameters from our params dictionary. A model is then trained and tested using a set of randomly sampled parameters. The goal of a Randomized Search is to explore a large set of possible hyperparameter spaces quickly — and the best way to accomplish this is via simple random sampling.

The code to perform a Randomized Search of hyperparameters for the k-NN algorithm below in the figure 27. It shows that the best n neighbors is 18 and the accuracy of the model has been **increased to 94.8%.**

Random Search for Hyper-Parameter Optimization

```
## Optimizing Hyper Parameter by Random Search Cross validation

from scipy.stats import randint as sp_randint
from sklearn.model_selection import RandomizedSearchCV
rf_params = {
    'n_neighbors': range(1,20),
}
n_iter_search=10
clf = KNeighborsClassifier()
Random = RandomizedSearchCV(clf, param_distributions=rf_params,n_iter=n_iter_search,cv=10,scoring='accuracy')
Random.fit(X_train, y_train)
print(Random.best_params_)
print("Accuracy:"+ str(Random.best_score_))
```

```
{'n_neighbors': 18}
Accuracy:0.9487531598005056
```

Fig 27: Random search for Hyper parameter Optimization.

GRID SEARCH HYPERPARAMETER:

The Grid Search tuning algorithm will train and evaluate a machine learning classifier methodically (and exhaustively) for each and every combination of hyperparameter values. The Grid Search algorithm's main advantage is also its main disadvantage: as an exhaustive search, the number of possible parameter values grows exponentially as the number of hyperparameters and hyperparameter values grows.

The code to perform a Grid Search of hyperparameters for the k-NN algorithm below in the figure 28. It shows that the best n neighbors is 18 and the accuracy of the model has been **increased to 94.86%.**

Optimizing KNN model using Grid Search CV

```
## ptimizing KNN model using Grid Search CV

from sklearn.model_selection import KFold, cross_val_score
from sklearn.model_selection import GridSearchCV
# Select an algorithm
algorithm = KNeighborsClassifier()
# Create 3 folds
seed = 13
kfold = KFold(n_splits=3, shuffle=True, random_state=seed)
# Define our candidate hyperparameters
hp_candidates = [{'n_neighbors': [15,18,20,16,17], 'weights': ['uniform','distance']}]
# Search for best hyperparameters
grid = GridSearchCV(estimator=algorithm, param_grid=hp_candidates, cv=kfold, scoring='accuracy')
grid.fit(X_train, y_train)
# Get the results
print(grid.best_score_)
print(grid.best_estimator_)
print(grid.best_params_)
```

```
0.9486985037917606
KNeighborsClassifier(n_neighbors=20)
{'n_neighbors': 20, 'weights': 'uniform'}
```

Fig 28: Random search for Hyper parameter Optimization.

H. PERFORMANCE:

In this project, we used a variety of performance metrics to determine the performance of the Machine Learning algorithms. Quality metrics related to classifications are presented here since the paper is genuinely concerned with classification issues. If the target variable is 1 (malignant) for packet attack prediction, it is a positive case, indicating that the packet has been attacked. And if the target variable is 0 (benign), then it is a negative instance, stating that the packet does not have been attacked.

CONFUSION MATRIX:

In summary, a method known as uncertainty matrix is used to evaluate the output of a classification algorithm. By comparing how many positive instances are correctly/incorrectly classified with how many negative instances are correctly/incorrectly classified, it is arguably the simplest way to control the efficiency of a classification model.

TRUE POSITIVES:

There are instances in which both the predictive and real class are accurate (1), i.e., where the packets has attacked and is therefore identified as having attacked packets by the model.

TRUE NEGATIVES:

True negatives arise when the expected class and the real class are both False (0), i.e. when a packets are benign and is also graded as such by the model.

FALSE NEGATIVES:

There are cases where the expected class is False (0) but the actual class is True (1), such as when the model classifies a packet as benign when they actually malicious.

TRUE NEGATIVES:

False positives arise when the expected class is True (1) but the actual class is False (0), for example, when a packet is identified as having problems by the model when they do not.

ACCURACY:

One of the metrics used to evaluate classification models is precision. The percentage of a forecast that is accurate is called accuracy. It calculates the proportion of correct predictions made by the model out of the total number of predictions made by the model. The formula of a accuracy is

$$\text{Accuracy} = (TP+TN) / (TP+TN+FP+FN)$$

RECALL:

It's a percentage of packets that are supposed to malicious among benign packets. The following formula can be used to measure recall:

$$\text{Recall} = TP / (TP+TN)$$

PRECISION:

It is described as a measure of proportion of packets that actually are malicious among those classified to be malicious by the model. The formula for Precision is as follows:

$$\text{Precision} = TP / (TP+FP)$$

F1 SCORE:

Weighted average of precision and recall is known as F1 score. Therefore false positives and false negatives are taken by this score into the consideration. Intuitively it is not as simple to grasp as accuracy, however F1 is typically additional helpful than accuracy. It is calculated as follows:

$$F1Score = (Precision * Recall) / (Precision + Recall) * 2$$

```
#confusion matrix of Logistic Regression
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

Predicted \ Actual	0	1
0	12258	7399
1	3530	165003

```
#confusion matrix for decision tree
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

Predicted \ Actual	0	1
0	13735	8314
1	2053	164088

```
#confusion matrix for SVM
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

Predicted \ Actual	0	1
0	4008	1320
1	440	20232

```
#confusion matrix of KNN
cm = pd.DataFrame(confusion_matrix(y_test, y_testpred).T)
cm.index.name = 'Predicted'
cm.columns.name = 'Actual'
cm
```

Predicted \ Actual	0	1
0	13053	7594
1	2700	164843

Confusion Matrix

LOGISTIC REGRESSION MODEL PERFORMANCE:

```
# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_test,y_testpred)*100)

#Test MSE
print ('Test MSE: ', mean_squared_error(y_test, y_testpred))


print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_test,y_testpred)))
```

```
[1 1 1 ... 1 1 0]
Accuracy is 94.19257133747809
Test MSE:  0.05807428662521919
Root_Mean_Squared_Error :  0.24098607143405446
```

```
#Classification report

print("Classification report :")
print(classification_report(y_test,y_testpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.62      0.78      0.69     15788
           1       0.98      0.96      0.97    172402

    accuracy                           0.94    188190
   macro avg       0.80      0.87      0.83    188190
weighted avg       0.95      0.94      0.94    188190
```

Fig 29: Performance metrics of logistic regression

DECISION TREE CLASSIFIER:

```
# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_test,y_testpred)*100)

#Test MSE
print ('Test MSE: ', mean_squared_error(y_test, y_testpred))


print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_test,y_testpred)))
```

```
[1 1 1 ... 1 1 0]
Accuracy is 94.49120569637068
Test MSE:  0.05508794303629311
Root_Mean_Squared_Error :  0.23470820828486827
```

```
#Classification report

print("Classification report :")
print(classification_report(y_test,y_testpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.62      0.87      0.73     15788
           1       0.99      0.95      0.97    172402

    accuracy                           0.94    188190
   macro avg       0.81      0.91      0.85    188190
weighted avg       0.96      0.94      0.95    188190
```

Fig 30: Performance metrics of decision tree classifier

SUPPORT VECTOR MACHINE:

```
# Predicting the Test set results
y_testpred = svm_model_linear.predict(X_test)
print(y_testpred)

# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_test,y_testpred)*100)

#Test MSE
print ('Test MSE: ', mean_squared_error(y_test, y_testpred))


print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_test,y_testpred)))
```

```
[0 1 1 ... 1 1 1]
Accuracy is 93.23076923076923
Test MSE:  0.06769230769230769
Root_Mean_Squared_Error :  0.26017745423519634
```

```
#Classification report

print("Classification report :")
print(classification_report(y_test,y_testpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.75      0.90      0.82      4448
           1       0.98      0.94      0.96     21552

    accuracy                           0.93     26000
   macro avg       0.87      0.92      0.89     26000
weighted avg       0.94      0.93      0.93     26000
```

Fig 31: Performance metrics of support vector machine

KNN ALGORITHM

```
# Predicting the Test set results
y_testpred = knn.predict(X_test)
print(y_testpred)

# model accuracy for the predicted value
from sklearn.metrics import accuracy_score
print ("Accuracy is",  accuracy_score(y_test,y_testpred)*100)

#Test MSE
print ('Test MSE: ', mean_squared_error(y_test, y_testpred))


print ('Root_Mean_Squared_Error : ', np.sqrt(metrics.mean_squared_error(y_test,y_testpred)))
```

```
[1 1 1 ... 1 1 1]
Accuracy is 94.52999628035495
Test MSE:  0.0547000371964504
Root_Mean_Squared_Error :  0.23388039079078518
```

```
#Classification report

print("Classification report :")
print(classification_report(y_test,y_testpred))
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.63      0.83      0.72     15753
           1       0.98      0.96      0.97    172437

    accuracy                           0.95    188190
   macro avg       0.81      0.89      0.84    188190
weighted avg       0.95      0.95      0.95    188190
```

Fig 32: Performance metrics of KNN classifier

PERFORMANCE ON TRAIN SET:

	ACCURACY	PRECISION	RECALL	F1 SCORE
LOGISTIC REGRESSION	94.24%	98%	96%	97%
DECISION TREE CLASSIFIER	95%	99%	95%	97%
KNN ALGORITHM	95.6%	99%	96%	98%
SVC	93.15%	98%	94%	96%

PERFORMANCE ON TEST SET:

	ACCURACY	PRECISION	RECALL	F1 SCORE
LOGISTIC REGRESSION	94.1%	98%	96%	97%
DECISION TREE CLASSIFIER	94.4%	99%	95%	97%
KNN ALGORITHM	94.6%	98%	96%	97%
SVC	93.2%	98%	94%	96%

I. CONCLUSION:

In terms of accuracy, Logistic regression have scored high figure without applying PCA. K-Neighbors is not far behind either. SVM scores less in accuracy. Decision Tree performs the worst among all. But executing without PCA take more time, hence it is not included in code. Application of PCA changes the accuracy of all the algorithms, K-Neighbors and Decision tree performs best after PCA is applied, even though there is a fall in accuracy. Considering the other performance matrix into account, a lot can be determined regarding the performance of the algorithms. Logistic regression per-forms better without the introduction of PCA, while KNN, Logistic Regression and Decision tree classifier per-forms better after PCA is applied to the dataset. KNN and Logistic Regression scores good when it comes to recall, which is vital in terms of packet attack prediction, after PCA is applied, even though there are declines in the values of all other performance metrics of both the mentioned algorithms. Keeping in mind that PCA reduces the run time exponential to huge extends in datasets (both small and large alike) and keeping the recall score into consideration, we can conclude that Logistic Regression, KNN and Decision tree with PCA performs better when it comes to Kitsune Network Attack – Mirai Dataset.

AA. REFERENCES:

[1] Nadia Chaabouni, Mohamed Mosbah , Akka Zemmari, Cyrille Sauvignac, and Parvez Faruki``Network Intrusion Detection for IoT Security Based on Learning Techniques" IEEE Communications Surveys \& Tutorials, Vol. 21, No. 3, Third Quarter 2019

[2] Ayush Kumar and Teng Joon Lim." EDIMA: Early Detection of IoT Malware Network Activity Using Machine Learning Techniques." 2019 IEEE 5th World Forum on Internet of Things (WF-IoT)

[3] Zhipeng Liu,Niraj Thapa,Addison Shaver,Kaushik Roy,Xiaohong Yuan and Sajad Khorsandroo ``Anomaly Detection on loT Network Intrusion Using Machine Learning"

[4] Shao-Chien Chen, Yi-Ruei Chen, and Wen-Guey Tzeng." Effective Botnet Detection Through Neural Networks on Convolutional Features." 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications.

[5] Rohan Doshi,Noah Apthorpe,Nick Feamster."Machine Learning DDoS Detection for Consumer Internet of Things Devices."2018 IEEE Symposium on Security and Privacy Workshops.

Premihaa_Project_Kitsune.docx

ORIGINALITY REPORT

45%	40%	10%	30%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

MATCHED SOURCE

2	towardsdatascience.com Internet Source	4%

1%

★ Submitted to University of Strathclyde

Student Paper

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		