Explainable ML for DDOS Attacks on Network Intrusion Detection System (NIDS)

Daksh Gopalani

Project Repository: https://github.com/DGopalani/NIDS

Background

A Network Intrusion Detection System (NIDS) is used to monitor network traffic and determine whether the traffic is malicious or benign. There are two main methods of filtering traffic: signature-based and anomaly-based detection. Signature-based detection relies on analyzing packet signatures and applying to them a long list of rules, which determine the packet's risk. These rules are based on known attacks, so signature-based detection is very useful in cases of previously employed attacks. Anomaly-based detection uses machine learning (ML) algorithms to analyze packet features and behavior and find and classify anomalous packets. The features of these anomalous packets are analyzed to determine if the packet is actually malicious or just benign. If the packet is found to be malicious, then the model determines which specific type of attack the packet should be classified as.[1]

Current Work

Nearly all modern day IDS's have an ML component for detecting and classifying anomalies. Although ML models are able to successfully identify anomalous packets through examining packet features, distinguishing between benign and malicious anomalies has proven to be a challenging task for these systems. Furthermore, once an AI system has been trained and has begun developing its own way of thinking, it becomes difficult to determine how the model reaches its decisions, since the decisions are based on the training data and complex algorithm. The lack of transparency in AI models' rationale is what has made explainable AI (XAI) a prevalent field. XAI is the process of explaining how and why an AI model reaches its decisions. This field is promising for NIDSs specifically, because it allows security analysts and businesses to understand how the ML model determined its classification. This information can then be used to determine whether a packet is truly malicious or benign and why. XAI can also be used by developers to improve the NIDS system, by helping them understand where the system is succeeding and where it is failing. XAI currently works by predicting accuracy, tracing decisions, and explaining decisions.[2]

Motivation

One issue with many NIDSs is that there is a very high false positive rate. Improving the AI model's accuracy is a pressing issue in cybersecurity, because it makes it difficult for security analysts to trust the information being presented by model.[3] One resolution to this problem is teaching a model to explain how it reached its result. Many NIDS systems today are unable to explain their decisions, because the ML algorithm is very

---

[1] Jacky. "What Is Network Intrusion Detection System (NIDS)?"
[2] "What Is Explainable AI (XAI)?." IBM.
[3] Jallad, Khloud Al, et al. "Anomaly Detection Optimization Using Big Data and Deep Learning to Reduce False-Positive - Journal of Big Data."

complex, making it difficult to pinpoint exactly how the algorithm reached its decision. My senior project aims to deliver as much information as possible to the user concerning how and why the model reached each of its decisions.

Contribution

For my senior project, I plan to build an Anomaly Based NIDS, which will explore different methods of explaining and visualizing how the ML aspect of the system reaches its decisions. My goals for this project are as follows:

1. Develop an Anomaly-Based NIDS system that works with relatively high accuracy.
2. Reveal how the ML model reaches its decisions step by step, based on the ML algorithm in place.
3. Visualize results so that they can be easily understood by security analysts.
4. Explain results for each packet so that they can be easily understood by security analysts.

Dataset in Use

The dataset I selected for this project is the "CIC-DDoS2019" DDOS Evaluation Dataset.[4] This dataset contains the abstract behavior of 25 users based on the HTTP, HTTPS, FTP, SSH, and email protocols.

Attack types include PortMap, NTP, SSDP, NetBIOS, WebDDoS, LDAP, MSSQL, UDP, UDP-Lag, SYN, NTP, DNS, TFTP and SNMP.

The data is spread over two days of attacks and each day includes network traffic in the form of raw data (PCAP files) and event logs (Windows and Ubuntu event log CSV files). Each CSV file has in it more than 80 network traffic features extracted from the raw data. AI techniques can be used to analyze the data in the CSV files. If I want to extract different features from the raw data than those supplied in the CSV files, I can run a different feature extractor on the raw data PCAP files.

The dataset includes the following file types:

- PCAP files: benign and the most up-to-date common DDoS attacks, which resemble true real-world data.
- CSV files: results of the network traffic analysis of the PCAP files using CICFlowMeter-V3 (feature extractor) with labeled flows based on the timestamp, source, and destination IPs, source and destination ports, protocols and attack.

[4] "Search UNB." *University of New Brunswick Est.1785.*

An informational paper included with the dataset also contains a table revealing which features are most valuable for identifying each type of attack. Since these features have already been found, I can use them without having to re-do the work of finding them in my own project.

DDoS Attacks

Distributed Denial-of-Service (DDoS) attacks occur when there is a flood of traffic to a server or web host. By creating enough traffic, an attacker can use up its target's bandwidth until it can no longer function. This results in a crash in the service being provided by the target machine.

DDoS attacks do not just occur from one IP address, so you can't just filter out an IP address to stop them. They are usually done with botnets, or groups of computers acting together, to overwhelm a server and bring it down.[5]

There are many different types of DDoS attacks. Given the time and resource constraints for this project, the scope of my NIDS system is limited to identifying the following types of attacks: UDP Flood, UDP-Lag, and SYN Flood. In UDP Flood attacks, the attack is set on a remote host by rapidly sending a large number of UDP packets to random ports of a target machine. This exhausts the bandwidth of the target machine causing the system to crash.[6] SYN Flood attacks, on the other hand, use up the system's resources by exploiting the TCP three-way handshake. In SYN Flood attacks, the host machine repeatedly sends SYN packets to the target machine until the server crashes.[7] UDP-Lag attacks are an odd gray area of DDoS attacks, which makes them a great test for the ML model's ability to identify the attacks. Rather than fully crashing the system, UDP-lag attacks aim to just significantly slow down the target machine, by delaying the target system's receival of network packets. These attacks are most common in online gaming platforms, where players only want to slow down the movement of opponents and not crash their entire game.. The attack is carried out by either using a hardware switch known as a lag switch, or by a software program that runs on the network and uses up the bandwidth of other users.[8]

Machine learning techniques are particularly useful in cases of anomalous traffic, where it is unclear if the feature data indicates an attack or not. A pre-trained ML model that can recognize various patterns in feature values will be able to more accurately and efficiently identify an anomalous network packet.

---

[5] "What Is a Distributed Denial-of-Service (Ddos) Attack?" Cloudflare.
[6] "UDP Flood Ddos Attack." Cloudflare.
[7] "SYN Flood Ddos Attack." Cloudflare.
[8] "Search UNB." University of New Brunswick Est.1785.

Implementing the Machine Learning Model

Training an ML model requires you to have pre-labeled training data and testing data. The model must be fed training data with labels that reveal what the correct evaluation of what the model should be. In the case of my project, each line of data represents a network packet. Each line of data has 80 feature values split by commas, followed by the line's label value. Examples of features include port, timestamp, and source IP Address. The final column of each line is the "Label" column which contains the value that the model should find for a given line. In my project's case, this column's value will be "BENIGN" if there is no attack to be found, "UDP" if a UDP-Flood attack should be found, "UDP-Lag" if a UDP-Lag attack is to be predicted, or "SYN" if a SYN Flood attack exists. When training the model with this training data, the label column's values are fed to the model so that the model can use its machine learning algorithm to figure out why a certain packet should be categorized as BENIGN, UDP, UDP-Lag, or SYN. When testing the model on fresh, unseen testing data, the label values are not given to the model. Instead, the model is made to predict what each packet should be identified as. Then, these predictions are compared with the actual label values of each packet in order to determine the model's overall accuracy.[9]

I chose to first implement an ML model that can recognize UDP-Lag attacks. My training data for detection of UDP-Lag attacks is stored in the file path, 'Data/01-12/UDPLag.csv', so I started by importing that file and building a pandas dataframe to hold all the data. A pandas dataframe is a data structure that contains two dimensional data with corresponding labels, similarly to a database. I then split the data frame into separate numeric and categorical dataframes.

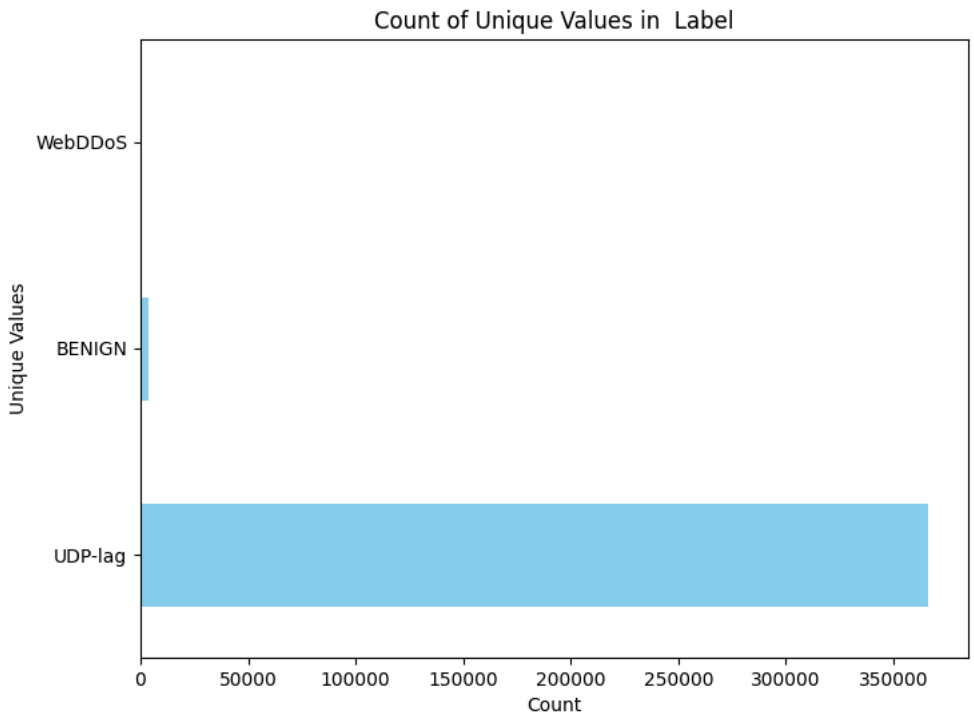
```
#Define the input CSV file path
csv_file_path = 'Data/01-12/UDPLag.csv'

df = pd.read_csv(csv_file_path)

# Select only numeric columns from the DataFrame
numeric_df = df.select_dtypes(include=['number'])


#df of categorical rows
# Filter the columns in df_subset that are not in numeric_df
categorical_columns = df.columns.difference(numeric_df.columns)
categorical_df = df[categorical_columns]
```

[9] "How to Train a Learning Model." Pluralsight.

For the next part of the project, I wanted to visualize the UDP-lag training data so that I could have a better understanding of its distribution. This visualization wasn't really important for the actual creation of the model and was more useful for my own knowledge.

I began by visualizing the 'Label' column, which holds the values that each packet should be identified as by the model.

```
#make bar graphs of categorical data
for column in categorical_df.columns:
    value_counts = categorical_df[column].value_counts()
    plt.figure(figsize=(8, 6))  # Adjust the figure size if needed
    # Plot the unique values on the y-axis and their counts on the x-axis
    value_counts.plot(kind='barh', color='skyblue')
    plt.xlabel('Count')
    plt.ylabel('Unique Values')
    plt.title(f'Count of Unique Values in {column}')
    plt.show()
```

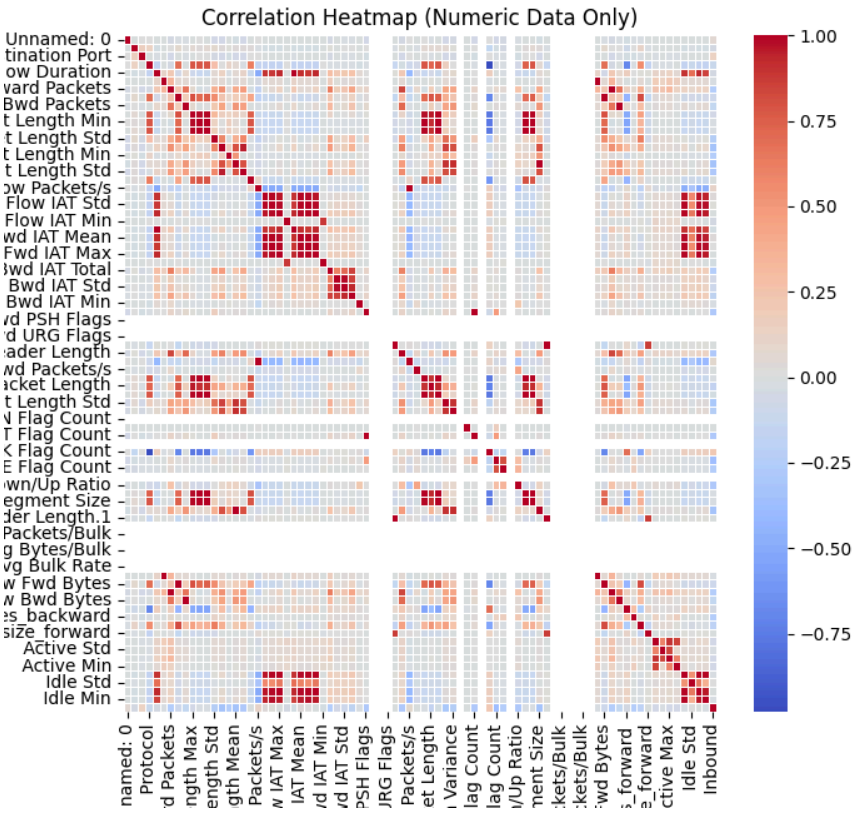


The results of this bar graph reveal that the UDP-lag training data only contains packets which are to be identified as "BENIGN", "UDP-Lag", and "WebDDoS". The "UDP-Lag" label indicates the packet is part of a UDP-Lag attack, the "BENIGN" label indicates that the packet is not part of any attack, and the "WebDDoS" label indicates a more general

type of DDoS Web attack, in which the attacker aims to disrupt the availability of a website or web application by overwhelming it with a flood of malicious traffic. There is not much specific information available in the dataset specifications regarding WebDDoS attacks, however, there are not many instances of these attacks present in the dataset as a whole, and this project's focus is not on my model's ability to detect WebDDoS attacks anyway. The results of this bar graph indicate that if a model is only trained on this UDP-Lag training data, then it will only be able to identify packets as either "BENIGN", "UDP-Lag", or "WebDDoS", since these are the only packets that the model will be exposed to.

Another visualization of the UDP-Lag training data that I was very interested in was the correlation between features (columns in the data), because this would help show which columns are related to one another. Two columns with a high correlation value indicate that they increase/decrease at a very similar rate and vice versa. A correlation matrix stores the correlation values between all columns of a dataset. To create the correlation matrix, I had to remove the columns with categorical data, because the pandas library does not allow you to create a correlation matrix with any categorical data columns. Categorical data columns are not allowed when creating a correlation matrix, because it's much more difficult to accurately compare qualitative data. Once I removed the categorical columns, I constructed a correlation matrix with the numeric columns. This correlation matrix was then used to make the heatmap, which shows the columns that are highly correlated. This visualization was important because it helped me validate which columns I should keep or remove. If two columns are highly correlated, then keeping both those columns may bias the model, since both columns' values have a very similar effect on the model's decisions.

```
#make correlation matrix
correlation_matrix = numeric_df.corr()

# Create a heatmap to visualize the correlation matrix
plt.figure(figsize=(8, 8))  # Adjust the figure size if needed
sns.heatmap(correlation_matrix, annot=False, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Heatmap (Numeric Data Only)")
plt.show()
```



I now have a correlation matrix that I could use to determine which features (columns) to keep/remove for the model. Generally, a correlation matrix is used to find which features are highly correlated and keep only one of those features. This is done to reduce the model's bias toward these highly correlated features. Currently, my dataset contains 80 features, many of which are irrelevant toward identifying DDoS attacks. Including all of these features will confuse my model since it will not know which features it should be focusing on to determine the correct label value. Therefore I need to test out different combinations of features and find the combination that results in the highest accuracy. Unfortunately, this process of tuning a model by including/removing different combinations of features can take a very long time and may even consume the entirety of my allotted time for this project. Luckily, the creators of the dataset I am using have already done this work and have included a chart in the dataset specifications indicating which features are best for identifying each type of attack. Below are the features relevant to identifying a UDP-Lag attack.

Name	Feature
UDP-lag	ACK Flag Count
	Init_Win_bytes_forward
	min_seg_size_forward
	Fwd IAT Mean
	Fwd IAT Max

This selection of features for identifying UDP-lag attacks makes sense. Here are explanations concerning why these features are relevant to UDP-Lag attacks:

- ACK Flag Count: The ACK Flag Count refers to the number of TCP packets with the ACK flag set in a given timeframe or within a specific connection session. UDP-Lag attacks often rely on exploiting vulnerabilities in the TCP handshake. In a legitimate TCP connection, the ACK flag should be set after the initial SYN-SYN/ACK exchange to confirm the connection establishment. However, in UDP-lag DDoS attacks, the ACK flag count might be low or nonexistent because the attacker does not complete the handshake process, leaving many half-open connections.[10]
- Init_Win_bytes_forward: This feature refers to the initial window size in bytes that a sender can send to a receiver without receiving an acknowledgment (ACK) flag in return. If many connections show a low value for the Init_Win_bytes_forward parameter, this could indicate suspicious activity. This is because a low initial window size suggests that the server is allocating resources for communication with clients but not receiving acknowledgments.[11]
- Min_seg_size_forward: This parameter refers to the minimum segment size observed in the forward direction of a connection. The segment size refers to the amount of data included in each network packet. Variations in segment sizes of packets may indicate anomalies or manipulation of packet sizes, contributing to the intentional slowing down of the network.
- Fwd IAT Mean: This parameter refers to Forward Inter-Arrival Time Mean, which measures the average time interval between consecutive forward packets. This feature's values can help identify patterns associated with intentionally spacing out UDP packets to cause lag.[12]
- FWD IAT Max: This parameter refers to the maximum Forward Inter-Arrival Time, which provides information about the longest time interval between consecutive forward packets. This feature is useful in detecting instances where the time gap between UDP packets is maximized to amplify the impact on network latency.

[10] "Understanding TCP Flags." Site24x7.
[11] Hindawi. "Table 3: Malware Detection in Self-Driving Vehicles Using Machine Learning Algorithms."
[12] "Welcome to Abertay University's Research Portal." Abertay University.

Please note that the main focus of this project is not finding out which features are most useful for identifying each type of DDoS attack and why. The most effective features for identifying each type of attack are already provided in the dataset's specifications, so there is no need for me to redo all this work. This project is instead more directed toward visualizing and explaining the decision-making process of the machine-learning model.

I next created a data structure called 'vars_to_keep' to hold the relevant features for UDP-Lag classification, which were discussed above. Training a model requires you to include what the result of each line should be. Therefore, I added the 'Label' column to the vars_to_keep data structure as well and applied feature mapping to the Label column to convert the Label values to integers. I converted the values of this column to integers because the ML Model library I am using, scikit-learn, requires the label/result values to be numeric. Finally I filtered the data so that only the columns included in the 'vars_to_keep' data structure are included. This means that for the case of the UDP-Lag model, the only columns kept in the data are 'ACK Flag Count', 'Init_Win_bytes_forward', 'min_seg_size_forward', 'Fwd IAT Mean', 'Fwd IAT Max', and 'Label'. I then used the 'vars_to_keep' list to filter all of the training data to only contain these columns. I did this filtering because I only want my model to be trained on the first five features, and I only want the model's results to be tested against the last column, which contains the values held in the 'Label' column.

```
#add Label to vars_to_keep
vars_to_keep.append(' Label')

#apply feature mappin on label column
df[' Label'] = df[' Label'].map({'BENIGN': 0, 'UDP-la

#make column filter list based on dataset info
vars_to_keep = [' ACK Flag Count', 'Init_Win_bytes_fo

#filter df based on filter list defined above
df_filtered = df[vars_to_keep]
```

Next I set up the testing dataset similarly to how I did above with the training dataset.

```
#test dataset
test_csv_file_path = 'Data/03-11/UDPLag.csv'
test_df = pd.read_csv(test_csv_file_path)

#apply feature mappin on label column
test_df[' Label'] = test_df[' Label'].map({'BENIGN': 0, 'UDPLag': 1, 'UDP': 2, 'Syn': 3, 'WebDDoS': 4})
#rename wrongly named column for consistency
test_df.rename(columns={'_bInit_Winytes_forward': 'Init_Win_bytes_forward'}, inplace=True)

#make testing filtered df based on vars_to_keep
test_df_filtered = test_df[vars_to_keep]
```

The next step of this implementation is preparing the training data. This requires splitting the data into two parts: one part that contains all the data that the model will use to identify the packet (values of relevant features), and another part that contains what each result should be (values in 'Label' column). In my code, the variable, X_train, contains the data to be fed to the model, and the variable, y_train, contains the results of each line of data from the 'Label' column. I prepared the testing data in the same way as I did the training data, into two variables, y_test and X_test.

```
#Prepare train data
y_train = df_filtered[[' Label']]
X_train = df_filtered.drop(columns=[' Label'], axis=1)
sc = MinMaxScaler()
X_train = sc.fit_transform(X_train)

#Prepare test data
y_test = test_df_filtered[[' Label']]
X_test = test_df_filtered.drop(columns=[' Label'], axis=1)
sc = MinMaxScaler()
X_test = sc.fit_transform(X_test)
```

I then successfully made sure there were no null values in any of the created training/testing datasets. This is an important step because null values will mess up the model's predictions and can cause the model to fail to compile.

```
#check for null values
print(np.isnan(y_test.values.any()))
print(np.isfinite(y_test.values.all()))
```

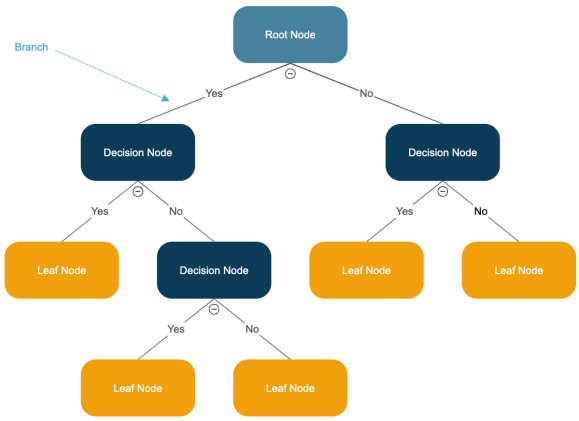
Next, I printed the shapes of all the datasets, and the datasets looked correctly organized. What I was mainly looking for here was for X_train and y_train to have the same number of lines and X_test and y_test to have the same number of lines. This check ensures that the number of packets for the model to compare against are consistent and a segmentation fault is avoided. I was also looking to ensure that X_train and X_test had 5 columns and y_train and y_test had 1 column. This check ensures that the model is only being fed the 5 input data columns and that the result variables only contained the single label column.

```
#print data shapes
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

```
(370605, 5) (725165, 5)
(370605, 1) (725165, 1)
```

Finally, I trained and tested the model with the Random Forest ML algorithm. I primarily chose to use a random forest because other implementations of NIDS ML models I found used this algorithm as well. This algorithm is generally chosen for these types of systems, because Random Forest is known for its accuracy in classification tasks. My model is being used to classify network traffic into various categories, so random forest is an effective choice.

To understand how random forest works, you first must understand another algorithm: decision trees. A decision tree is a structure where each internal node represents a "test" on a feature, each branch represents the outcome of the test, and each leaf node represents a class label or a value. Decision trees are built recursively by splitting the data based on the most informative features at each step.



Random forest implements a ‘forest’ of decision trees. While training each decision tree in the random forest, a random subset of features is selected as candidates for each split.

The randomness of the selection helps decorrelate the trees and reduce overfitting, since it ensures each tree is trained on different subsets of features.

Once all the decision trees are trained, each tree makes its own predictions individually. Each tree then "votes" for a class label, and the class with the most votes is chosen as the final prediction. These votes can also be made visible, which can be insightful to how a model reached its decision. The final prediction shown by default is just the class with the most “votes”. Random Forest has more accurate results than decision trees because it combines the results of multiple different decision trees, reducing overfitting and sensitivity to outliers.[13]

```
#Train on Random Forest Model
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

clfr = RandomForestClassifier(n_estimators = 5)
start_time = time.time()
clfr.fit(X_train, y_train.values.ravel())
end_time = time.time()
print("Training time: ", end_time-start_time)

#Test on Random Forest Model
start_time = time.time()
y_test_pred = clfr.predict(X_train)
end_time = time.time()
print("Testing time: ", end_time-start_time)

#Print results
print("Train score is:", clfr.score(X_train, y_train))
print("Test score is:", clfr.score(X_test, y_test))
```

The results for the training dataset were excellent, however, the initial results for the testing dataset were awful, as can be seen below. The training data accuracy was 0.995, whereas the testing accuracy was 0.006. I was hoping to have a testing score much closer to the training score.

[13] “Random Forest: A Complete Guide for Machine Learning.” Built In.

```
Training time:  0.4378190040588379
Testing time:  0.082180023193359 38
Train score is: 0.995251008486124
Test score is: 0.00626891810829 25954
```

Upon further investigation regarding why the testing score was so low, I realized it was because the testing dataset exposes the model to UDP and SYN attacks in addition to UDP-lag, WebDDoS, and BENIGN traffic. The model was only trained to recognize UDP-Lag, WebDDoS, and BENIGN traffic, since these are the only labels present in the training data. This means that the model is basically failing to identify UDP and SYN traffic in the testing data.

In order to account for this issue, I filtered the testing data frame so that it only held columns with a Label value of Benign (0), UDP-Lag (1), or WebDDoS (4) with the following code:

```
#keep only 0, 1, and 4 in Label column of test_df_filtered
test_df_filtered = test_df_filtered[test_df_filtered[' Label'].isin([0, 1, 4])]
```

This filtering dramatically improved the results of the ML Random Forest algorithm implementation.

```
Random Forest Classifier
Training time:  0.39696621894836426
Testing time:  0.07332372665405273
Train score is: 0.9953805264365024
Test score is: 0.7502104022891769
```

Although random forest is the algorithm that is most commonly used in these systems, I decided to still test with the Guassian Naive Bayes algorithm to compare with the Random Forest results. It performed much worse. This makes sense because Guassian Naive Bayes assumes that all features are independent of each other and that features follow a normal distribution.[14] This is definitely not the case here because many of the features are highly correlated. Here is the code for the Naive Bayes implementation and its results:

[14] S, Lavanya. “Gaussian Naive Bayes Algorithm for Credit Risk Modelling.” Analytics Vidhya

```
print("Gaussian Naive Bayes Classifier")
# Gaussian Naive Bayes
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

clfg = GaussianNB()
start_time = time.time()
clfg.fit(X_train, y_train.values.ravel())
end_time = time.time()
print("Training time: ", end_time-start_time)

start_time = time.time()
y_test_pred = clfg.predict(X_train)
end_time = time.time()
print("Testing time: ", end_time-start_time)

print("Train score is:", clfg.score(X_train, y_train))
print("Test score is:", clfg.score(X_test, y_test))
```

```
Gaussian Naive Bayes Classifier
Training time:  0.046190023422241 21
Testing time:  0.029289960861206055
Train score is: 0.8607735999244479
Test score is: 0.3179599394041407
```

I then implemented the Decision Tree algorithm to compare with the other ML implementations. It performed better than Naive Bayes but worse than Random Forest, which makes sense based on my description of how random forest works above. Here is the code and results:

```
# Decision Tree
print("Decision Tree Classifier")
from sklearn.tree import DecisionTreeClassifier
clfd = DecisionTreeClassifier(criterion ="entropy", max_depth = 4)
start_time = time.time()
clfd.fit(X_train, y_train.values.ravel())
end_time = time.time()
print("Training time: ", end_time-start_time)
start_time = time.time()
y_test_pred = clfd.predict(X_train)
end_time = time.time()
print("Testing time: ", end_time-start_time)
print("Train score is:", clfd.score(X_train, y_train))
print("Test score is:", clfd.score(X_test, y_test))
```

```
Decision Tree Classifier
Training time:  0.1373138427734375
Testing time:  0.013846874237060547
Train score is: 0.994622306768662
Test score is: 0.6349099478202322
```

Industry standards for a ML model testing score is between 70% and 90% and anywhere above 70% is considered acceptable as realistic and valuable output.[15] I decided to continue with the Random Forest ML algorithm over the others.

I next trained a separate model to only recognize UDP data, very similarly to the model I created above to recognize UDP-lag attacks. First I prepared the UDP training data.

```
#Prepare UDP train data
UDP_y_train = UDP_df_filtered[[' Label']]
UDP_X_train = UDP_df_filtered.drop(columns=[' Label'], axis=1)
UDP_sc = MinMaxScaler()
UDP_X_train = UDP_sc.fit_transform(UDP_X_train)
```

Then I filtered the UDP testing dataset so that it kept only values that are labeled “BENIGN”, “UDP”, and ‘WEBDDoS”. I did this because these are the only labels present in the UDP training dataset, so these are the only labels that the model should be able to recognize.

[15] “What Is a Good Accuracy Score in Machine Learning?” Deepchecks.

```
#apply feature mappin on label column
test_df[' Label'] = test_df[' Label'].map({'BENIGN': 0, 'UDPLag': 1, 'UDP': 2, 'Syn': 3, 'WebDDoS': 4})
#rename wrongly named column for consistency
test_df.rename(columns={'_bInit_Winytes_forward': 'Init_Win_bytes_forward'}, inplace=True)

#make testing filtered df based on vars_to_keep
test_df_filtered = test_df[UDP_vars_to_keep]

#keep only 0, 1, and 4 in Label column of test_df_filtered
test_df_filtered = test_df_filtered[test_df_filtered[' Label'].isin([0, 2, 4])]
```

Next I trained the model on the UDP data, just like I did above with the UDP-Lag model.

```
#Train on UDP data
clfr = RandomForestClassifier(n_estimators = 5)
start_time = time.time()
clfr.fit(UDP_X_train, UDP_y_train.values.ravel())
end_time = time.time()
print("Training time: ", end_time-start_time)
```

Then I tested the model with the UDP data.

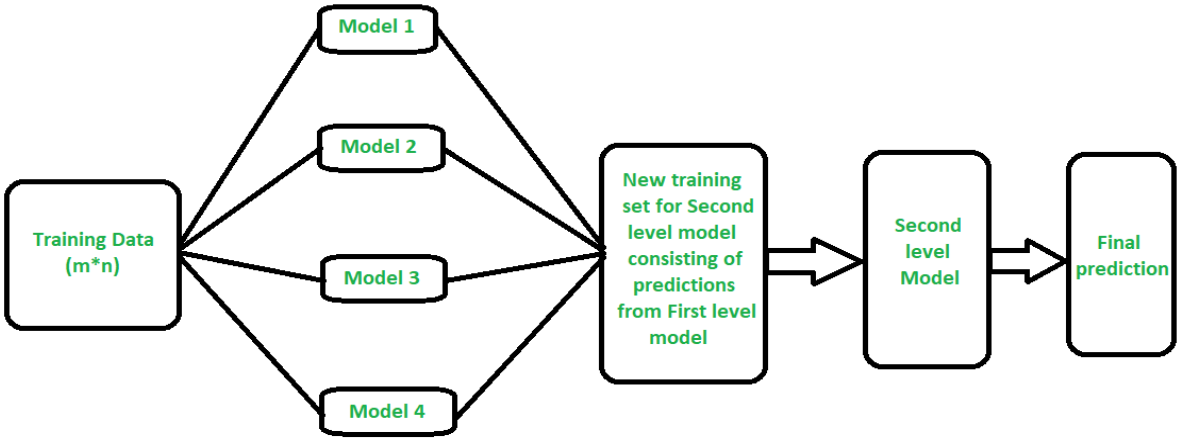
```
#Test on Random Forest Model with UDP data
start_time = time.time()
y_test_pred = clfr.predict(X_test)
end_time = time.time()
print("Testing time: ", end_time-start_time)

#Print results
print("Train score is:", clfr.score(UDP_X_train, UDP_y_train))
print("Test score is:", clfr.score(X_test, y_test))
#micro=Calculate metrics globally by counting the total true positives, false negatives and false positives.
precision = precision_score(y_test, y_test_pred, average='micro')
recall = recall_score(y_test, y_test_pred, average='micro')
conf_matrix = confusion_matrix(y_test, y_test_pred)
false_positive_rate = conf_matrix[0, 1] / (conf_matrix[0, 0] + conf_matrix[0, 1])
print("Precision:", precision)
print("Recall:", recall)
print("False Positive Rate:", false_positive_rate)
print("Confusion Matrix:")
print(conf_matrix)
```

The model performed well and the results were as follows.

```
Training time:  5.3509650230407715
Testing time:  0.025605201721191406
Train score is: 0.999926358118874
Test score is: 0.9922260453223274
Precision: 0.9922260453223274
Recall: 0.9922260453223274
False Positive Rate: 0.22000983284169126
Confusion Matrix:
[[  3173    895]
 [    11 112464]]
```

At this point of the project I have two separate models: one for UDP-Lag attack identification and one for UDP attack Identification. The next step is to combine these models into one combined model that can identify and differentiate between benign traffic, UDP attacks, UDP-Lag attacks, and WebDDoS attacks.

This combination of models can be done with a method known as stacking, which allows for ensembling multiple models. Stacking works by first training base models. In my case the base models are the UDP-Lag model and the UDP model. Then the predictions on the validation set are used as features to train the combined model. The combined model learns to combine the predictions of the base models into a final combined prediction. Here is a visualization of the process of stacking.[16]



This new combined model would have to be trained and tested on data that includes instances of UDP and UDP-Lag packets. For this reason, I combined the UDP-Lag and UDP training datasets for the combined model's training dataset.

I then implemented stacking to combine my models for UDP and UDP-Lag. I then set the features in use by this model to include both features used by the UDP and UDP-Lag models. The code is as follows.

[16] GfG. "Stacking in Machine Learning." GeeksforGeeks.

```
#Overall training dataset - contains UDP and UDPLag
#combine UDP_df_filtered and UDPLag_df_filtered
frames = [UDP_df, UDPLag_df]
#set vars to keep to UDP_vars_to_keep + UDPLag_vars_to_keep - Label
vars_to_keep = UDP_vars_to_keep + UDPLag_vars_to_keep[:-1]
df = pd.concat(frames)
#filter df based on filter list defined above
df_filtered = df[vars_to_keep]
```

For the testing dataset of this combined model, I used the UDP-Lag dataset, and included all packets labeled as “BENIGN”, “UDP”, “UDP-Lag”, and “WEBDDoS”, since the combined model should be able to identify all of these packets. The code for the test data is as follows:

```
#Overall test dataset - contains UDP and UDPLag
test_csv_file_path = 'Data/03-11/UDPLag.csv'
test_df = pd.read_csv(test_csv_file_path)
#apply feature mappin on label column
test_df[' Label'] = test_df[' Label'].map({'BENIGN': 0, 'UDPLag': 1, 'UDP': 2, 'Syn': 3, 'WebDDoS': 4})
#rename wrongly named column for consistency
test_df.rename(columns={'_bInit_Winytes_forward': 'Init_Win_bytes_forward'}, inplace=True)
#make testing filtered df based on vars_to_keep
test_df_filtered = test_df[vars_to_keep]
#keep only 0, 1, and 4 in Label column of test_df_filtered
test_df_filtered = test_df_filtered[test_df_filtered[' Label'].isin([0, 1, 2, 4])]
```

In order to combine the UDP and UDP-Lag models I first had to set each model to a variable. To do this I did the following:

- After setting up the training sets for the UDP-Lag model, I set the UDP-Lag model to a variable according to this training data. Here is the code:

```
#Prepare UDPLag train data
UDPLag_y_train = UDPLag_df_filtered[[' Label']]
UDPLag_X_train = UDPLag_df_filtered.drop(columns=[' Label'], axis=1)
UDPLag_sc = MinMaxScaler()
UDPLag_X_train = UDPLag_sc.fit_transform(UDPLag_X_train)

#Setup UDPLag Model
model_UDPLag = make_pipeline(MinMaxScaler(), RandomForestClassifier(n_estimators=5))
model_UDPLag.fit(UDPLag_X_train, UDPLag_y_train.values.ravel())
```

- I also set the UDP model to a variable as seen below:

```
#Prepare UDP train data
UDP_y_train = UDP_df_filtered[[' Label']]
UDP_X_train = UDP_df_filtered.drop(columns=[' Label'], axis=1)
UDP_sc = MinMaxScaler()
UDP_X_train = UDP_sc.fit_transform(UDP_X_train)

#Setup UDP Model
model_UDP = make_pipeline(MinMaxScaler(), RandomForestClassifier(n_estimators=5))
model_UDP.fit(UDP_X_train, UDP_y_train.values.ravel())
```

Then I prepared the training and testing data for the combined model as follows:

```
#prepare overall UDP and UDPlag train data
y_train = df_filtered[[' Label']]
X_train = df_filtered.drop(columns=[' Label'], axis=1)
sc = MinMaxScaler()
X_train = sc.fit_transform(X_train)

#Prepare overall UDP and UDPLag test data
test_y = test_df_filtered[[' Label']]
test_X = test_df_filtered.drop(columns=[' Label'], axis=1)
#test_sc = MinMaxScaler()
test_X = sc.fit_transform(test_X)
```

Finally, I wrote the code for the combined model, which implemented stacking of the UDP and UDP-Lag models.

```
#Train on Stacking Model
# Combine models using a meta-model (Stacking)
combination = StackingClassifier(estimators=[('UDP', model_UDP), ('UDPLag', model_UDPLag)])
combination.fit(X_train, y_train.values.ravel())

# Test the meta-model on the combined test data
y_test_pred_combined = combination.predict(test_X)

print("Length of y_test_pred_combined:", len(y_test_pred_combined))
print("Length of test_y:", len(test_y))

# Evaluate the combined model
combined_accuracy = accuracy_score(test_y, y_test_pred_combined)
combined_precision = precision_score(test_y, y_test_pred_combined, average='micro')
combined_recall = recall_score(test_y, y_test_pred_combined, average='micro')
combined_conf_matrix = confusion_matrix(test_y, y_test_pred_combined)

print("Combined Model Metrics:")
print("Combined Accuracy:", combined_accuracy)
print("Combined Precision:", combined_precision)
print("Combined Recall:", combined_recall)
print("Combined Confusion Matrix:")
print(combined_conf_matrix)
```

The results of this combined model were highly successful:

```
Combined Model Metrics:
Combined Accuracy: 0.9534437913795433
Combined Precision: 0.9534437913795433
Combined Recall: 0.9534437913795433
Combined Confusion Matrix:
[[  3629    139    297      3]
 [    15     90   1768      0]
 [     7   3284 109184      0]
 [     0      0      0      0]]
```

At this point of the project, the combined model is not accounting for Syn Flood attacks, which account for a large portion of the attacks in the test file. My testing dataset is currently filtering out the Syn attacks in the test dataset, since I have not yet trained the combined model to recognize Syn Attacks. The next step is to create and train a Syn model, similarly to how I created and trained the UDP-lag and UDP models.

```
#SYN training dataset
SYN_csv_file_path = 'Data/01-12/Syn.csv'
SYN_df = pd.read_csv(SYN_csv_file_path)
SYN_df[' Label'] = SYN_df[' Label'].map({'BENIGN': 0, 'UDP-lag': 1, 'DrDoS_UDP': 2, 'Syn': 3, 'WebDDoS': 4, 'DrDoS_MSSQL': 5})
SYN_vars_to_keep = [' ACK Flag Count', 'Init_Win_bytes_forward', ' min_seg_size_forward', 'Fwd IAT Total', ' Flow Duration', ' Label']
SYN_df_filtered = SYN_df[SYN_vars_to_keep]

#SYN test dataset
SYN_test_csv_file_path = 'Data/03-11/Syn.csv'
SYN_test_df = pd.read_csv(SYN_test_csv_file_path)

#apply feature mappin on label column
SYN_test_df[' Label'] = SYN_test_df[' Label'].map({'BENIGN': 0, 'UDPLag': 1, 'UDP': 2, 'Syn': 3, 'WebDDoS': 4, 'MSSQL': 5, 'LDAP': 6})
#make testing filtered df based on vars_to_keep
SYN_test_df_filtered = SYN_test_df[SYN_vars_to_keep]
#keep only 0 and 3 in Label column of test_df_filtered
SYN_test_df_filtered = SYN_test_df_filtered[SYN_test_df_filtered[' Label'].isin([0, 3])]
```

```
print('')
print("SYN")
#Prepare SYN train data
SYN_y_train = SYN_df_filtered[[' Label']]
SYN_X_train = SYN_df_filtered.drop(columns=[' Label'], axis=1)
SYN_sc = MinMaxScaler()
SYN_X_train = SYN_sc.fit_transform(SYN_X_train)

#Prepare SYN test data
SYN_y_test = SYN_test_df_filtered[[' Label']]
SYN_X_test = SYN_test_df_filtered.drop(columns=[' Label'], axis=1)
SYN_sc = MinMaxScaler()
SYN_X_test = SYN_sc.fit_transform(SYN_X_test)

#print data shapes
print(SYN_X_train.shape, SYN_X_test.shape)
print(SYN_y_train.shape, SYN_y_test.shape)
```

I then assigned a variable to the Syn model:

```
#Setup SYN Model
model_SYN = make_pipeline(MinMaxScaler(), RandomForestClassifier(n_estimators=5))
model_SYN.fit(SYN.SYN_X_train, SYN.SYN_y_train.values.ravel())
```

Next I alternated the overall stacked model's training set to include the SYN training data:

```
#Overall training dataset - contains UDP and UDPLag and SYN data
frames = [UDP.UDP_df, UDPLag.UDPLag_df, SYN.SYN_df]
#set vars to keep to UDP_vars_to_keep + UDPLag_vars_to_keep - Label
vars_to_keep = UDP.UDP_vars_to_keep + UDPLag.UDPLag_vars_to_keep[:-1] + SYN.SYN_vars_to_keep[:-1]
vars_to_keep = list(set(vars_to_keep))
#filter df based on filter list defined above
df = pd.concat(frames)
df_filtered = df[vars_to_keep]
```

After this, I altered the overall testing data to include attacks labeled as "Syn":

```
#Overall test dataset
test_df = pd.read_csv('Data/03-11/UDPLag.csv')
#apply feature mapping on label column
test_df[' Label'] = test_df[' Label'].map({'BENIGN': 0, 'UDPLag': 1, 'UDP': 2, 'Syn': 3, 'WebDDoS': 4, 'MSSQL': 5})
#rename wrongly named column for consistency
test_df.rename(columns={'_bInit_Winytes_forward': 'Init_Win_bytes_forward'}, inplace=True)
#make testing filtered df based on vars_to_keep
test_df_filtered = test_df[vars_to_keep]
#keep only 0, 1, and 4 in Label column of test_df_filtered
test_df_filtered = test_df_filtered[test_df_filtered[' Label'].isin([0, 1, 2, 3, 4, 5])]

print(test_df_filtered[' Label'].value_counts())
```

I then altered the combined model to include the Syn model in its stacking:

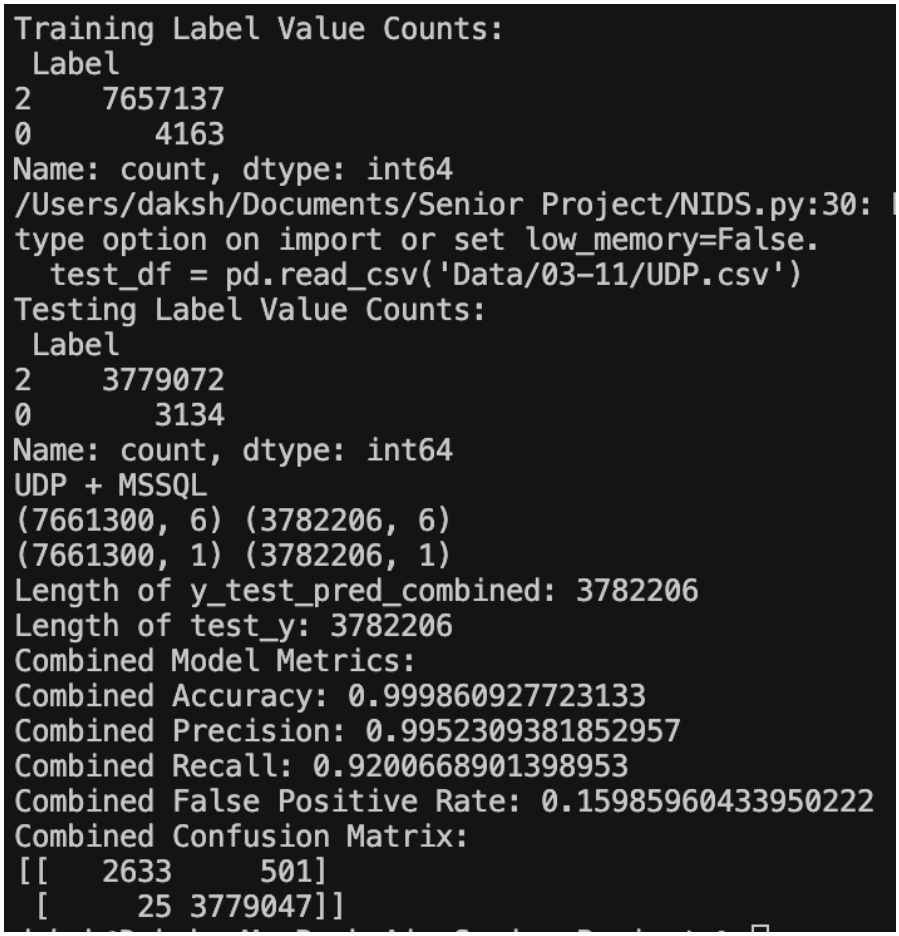
```
# Combine models using a meta-model (Stacking)
combination = StackingClassifier(estimators=[
    ('UDP', model_UDP),
    ('UDPLag', model_UDPLag),
    ('SYN', model_SYN)])
combination.fit(X_train, y_train.values.ravel())
```

Finally, I tested on testing data that included packets labeled as "UDP", "UDP-Lag", "SYN", "BENIGN", and "WEBDDoS". The results were successful with high accuracy, as can be seen below.

```
UDPLag + UDP + SYN
(5090088, 11) (725165, 11)
(5090088, 1) (725165, 1)
Length of y_test_pred_combined: 725165
Length of test_y: 725165
Combined Model Metrics:
Combined Accuracy: 0.9715306171698854
Combined Precision: 0.5963766988128523
Combined Recall: 0.5832103545275733
Combined Confusion Matrix:
[[  3712    114    198     32     12]
 [    11    104   1757      1      0]
 [     7   2952 109515      1      0]
 [     2  15474     84 591189      0]
 [     0      0      0      0      0]]
```

I next decided to incorporate MSSQL attacks into this combined model, however, this caused me to run into trouble. When including the MSSQL model, the accuracy and precision of the combined model dropped significantly. To figure out why this was happening, I began testing with different combinations of models. For example, I tried stacking only the UDP-Lag and MSSQL models, only the UDP and MSSQL models, and only the SYN and MSSQL models.

I began my testing by including only UDP and MSSQL attacks. I started by assigning all UDP and MSSQL labels in the training and testing data to the same value of 2. By doing this, I am basically able to see if the model is able to differentiate attacks from benign traffic. Since UDP and MSSQL are set to the same value of 2, if the model labels any packet as 2, this means it found an attack and if it labels any packet as 0, it means it found the packet to be benign. The result of this test can be used to determine if the combined model is confusing benign and malicious attacks or UDP and MSSQL attacks. The results are as follows:



The high accuracy means that the combined model is successfully able to distinguish between benign and malicious attacks. This interpretation can be made because I have set all attack packets to the same value of 2 and all benign traffic is set to the same value of 0. The model is therefore only being tested on whether it identifies a packet as an attack or benign. Since this test was successful, it can be concluded that the model is good at telling apart malicious from benign traffic and the low accuracy from introducing MSSQL attacks into the combined model is coming from the combined model struggling to tell MSSQL and UDP attacks apart. The model works fine when either UDP or MSSQL attacks are included, but not both. The model is still able to predict that there is an attack but incorrectly identifies the type of attack.

As a potential solution to this problem, I found that I can print out the model's percent likelihood that a packet belongs to one class versus another. If the model finds that a packet has a 49% chance of being UDP and 51% chance of being MSSQL, the packet is still classified as MSSQL and that is what the user currently sees. Since the classifications are so close, it would be useful for a security analyst to see the values of the likelihood that a packet belongs to one class versus another, so that they can

investigate whether the packet is really a MSSQL or UDP attack. Therefore I created the following code to print out the likelihoods per class for each packet.

```python
#Print likelihood of each class
print("Likelihood of each class:")
probabilities = combination.predict_proba(test_X)

# Print the probabilities for the first few samples
for i in range(min(500, len(test_X))):
    #formatted_probs = [f"{prob:.2f}" for prob in probs]
    #print(formatted_probs)
    print(f"Test Instance {i + 1}:")
    print(f"Features: {test_X[i]}")

    # If your model supports feature importances (e.g., RandomForest), you can print them
    if hasattr(combination, 'feature_importances_'):
        print(f"Feature Importances: {combination.feature_importances_}")

    # Print predicted probabilities for each class
    print("Predicted Probabilities:")
    for j, prob in enumerate(probabilities[i]):
        print(f"Class {j}: {prob:.4f}")

    print("\n")
```

Here were the results:

```
Likelihood of each class:
Test Instance 1:
Features: [0.5        0.         0.24309148 1.         0.99999927 0.        ]
Predicted Probabilities:
Class 0: 0.0000
Class 1: 0.2559
Class 2: 0.7441


Test Instance 2:
Features: [0.         0.         0.45772488 1.         0.99999927 0.        ]
Predicted Probabilities:
Class 0: 0.0000
Class 1: 0.2426
Class 2: 0.7574


Test Instance 3:
Features: [0.5        0.         0.45604639 1.         0.99999927 0.        ]
Predicted Probabilities:
```

Explaining/Visualizing Results

An incredibly important aspect of using artificial intelligence in a field as critical as cybersecurity is explaining the results to security analysts.[17] I realized the significance of these explanations through my experience working with CrowdStrike, a cybersecurity firewall tool. The company's software uses machine learning to classify the severity of vulnerabilities on host devices. Although their model is proven to be quite accurate, one major drawback I discovered when using this service was the lack of explanation of how the severity score is found. When I asked their team whether they could explain how they obtained their severity score, they said the score is determined by their internal machine learning model, which does not have any means to explain its decision-making process. This issue makes it more difficult to know why something is classified as a vulnerability, which can be vital information, especially in cases of false positives. Knowing how a severity score has been found can be very insightful to determine what vulnerabilities should be prioritized and why.

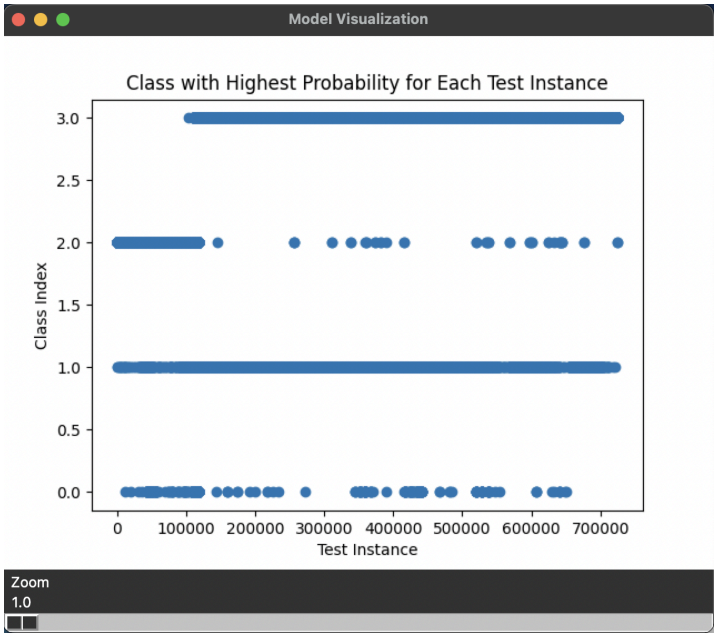
Although my model works at a much smaller scope and scale as that of CrowdStrike, I aim to visualize to a security analyst, who is using my software, why a packet is classified as UDP, UDP-lag, BENIGN, or SYN. CrowdStrike surely has a more well-researched and well-implemented model than my own, however, my research can eventually be applied to larger scale models, such as the one employed at CrowdStrike, in order to increase the transparency of the software's inner mechanisms.

If I was a security analyst studying packets for attacks, I would be mainly interested in the features used for the classification of a packet. I would like to know what these features are, what the values of these features are, and why these features are relevant to a packet's classification. This information could help show me whether the model's classification is valid and justified. I would also want to see the values of more general features such as Timestamp, Source Port, Destination Port, Source IP Address, and Destination IP Address. These general features could help me discover if a potentially malicious packet was sent/received around the same time as other similarly labeled packets, and whether this packet has matching Source/Destination IP addresses or ports as other similarly labeled packets. As a security analyst, I would also be interested in seeing the model's breakdown of the percent likelihood that the packet belongs to one class versus another (EX: 50% UDP, 40% MSSQL, 10% BENIGN). This information can be incredibly useful in cases where the classification is very close between two types of attacks and it could really be either one.

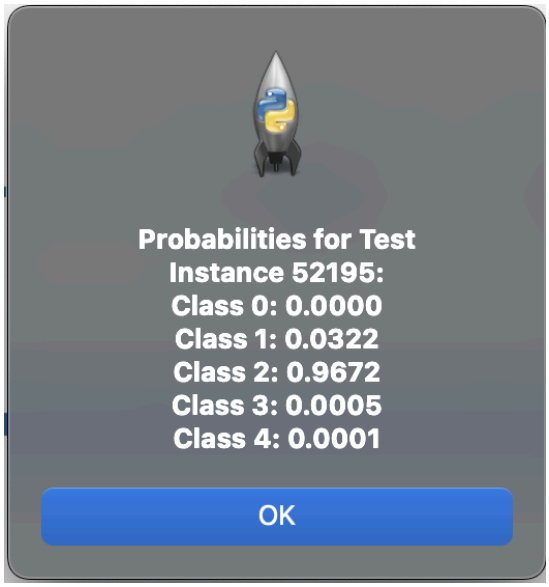
---

[17] Neupane, Subash, et al. "Explainable Intrusion Detection Systems (X-IDS): A Survey of Current Methods, Challenges, and Opportunities."

My initial direction for the GUI was to plot the predicted classes for all the data. This is what my initial GUI looked like. Each class index represents a different classification, with 0 indicating BENIGN, 1 indicating UDP, 2 indicating UDP-Lag, and 3 indicating Syn.

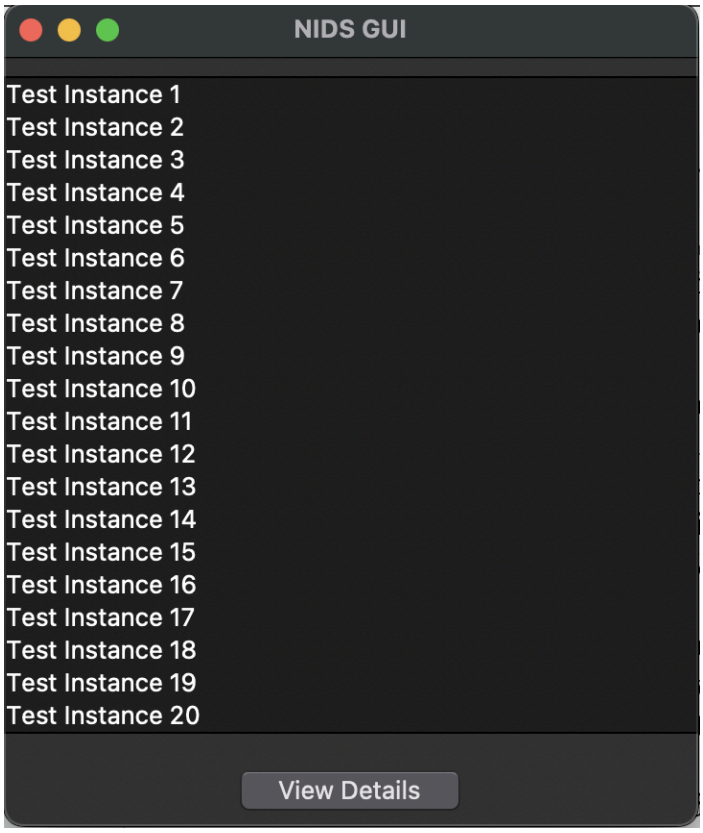


Upon clicking a point, you can see that line's predicted class breakdown from the model.

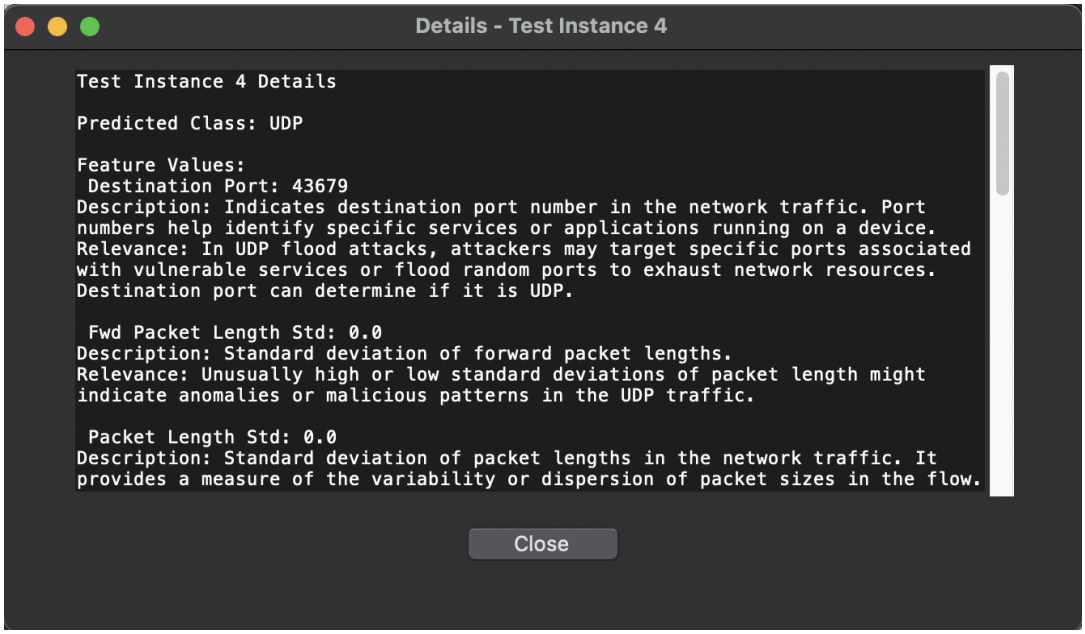


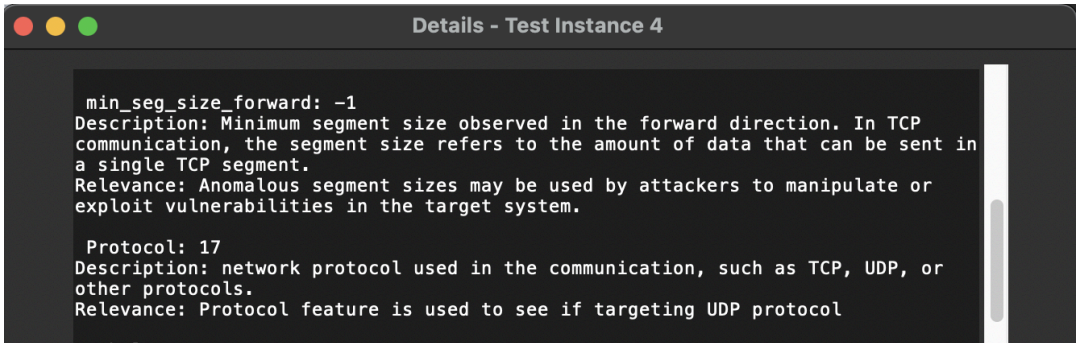
As you can see, this GUI shows too many test instances at once, and this representation of the data is not really useful for finding a specific attack and examining it. Therefore, I

decided to change my approach and show each line of network traffic on its own row as a test instance as such:

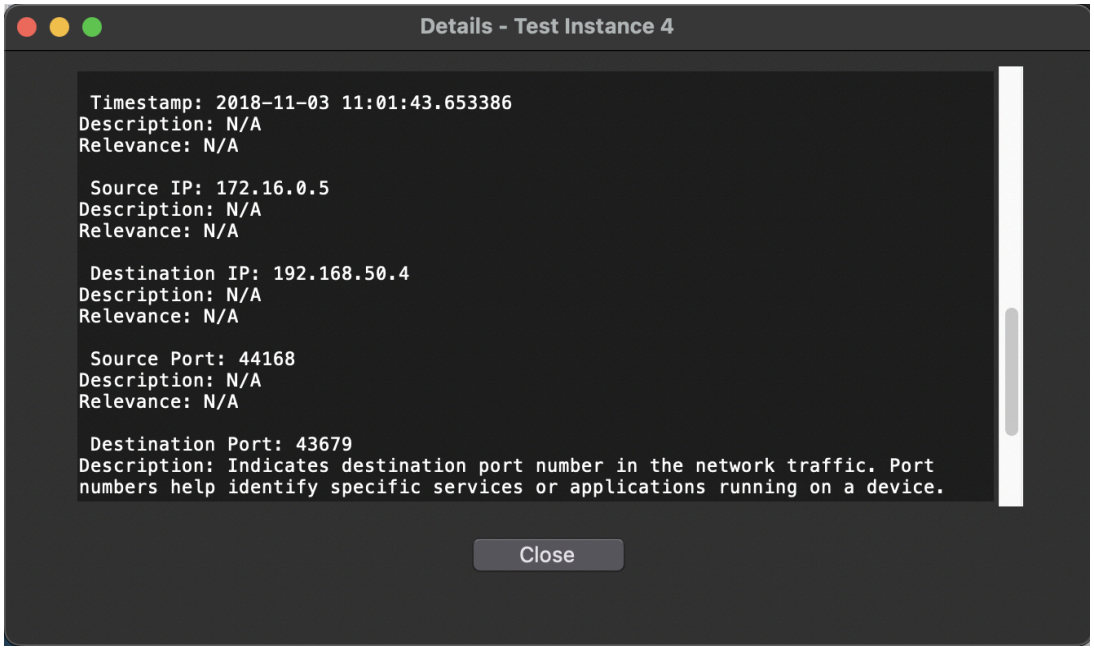


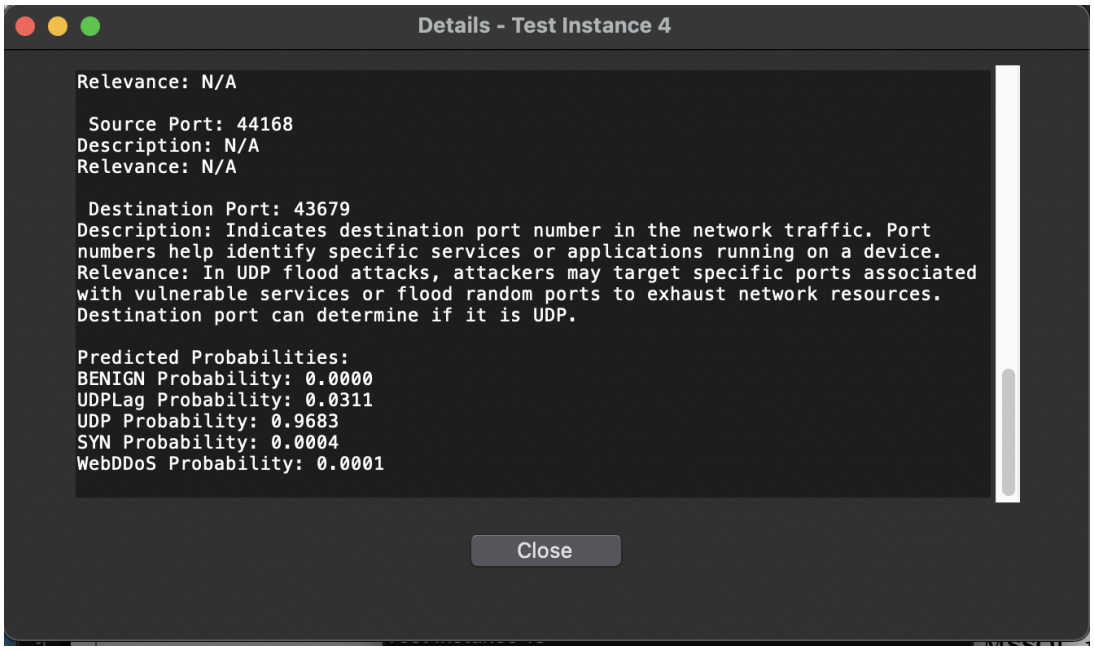
Upon clicking on a test instance, you can see the value of the assigned class (UDP, UDP-Lag, SYN, or BENIGN). Then based on this assigned class, you can see all the network features that are relevant to this classification, each feature's value, a description of what each feature means, and reasoning behind each feature's relevance to the final classification. All this information can be used by a security analyst to validate whether the classification is accurate. You can see the GUI below.





Upon scrolling further down, you can also see features, which were not relevant to the model's classification, but which are useful for security analysts to find a potential attack. These features include Timestamp, Source IP, Destination IP, Source Port, and Destination Port. These features are more useful for comparing packets against other similar packets, as was discussed above.





At the end, I have also included a breakdown of the model's percent likelihoods that the current line of network traffic is to be classified as one class versus another. In this test instance, the model found that the attack has a 96.53% chance of being a UDP Flood attack, a 3.11% chance of being a UDPLag attack, a 0.04% chance of being a SYN Flood attack, and a 0.01% chance of being a WebDDoS attack. This breakdown is useful in scenarios where the likelihood of multiple attacks is very close and determining the actual label value is confusing. From my research, I found that certain attacks have very similar signs in terms of network traffic features, and often even humans cannot tell

them apart by just looking at the feature values. With this score breakdown and explanations of features, security analysts can find these confusing cases and further their investigation to figure out the real classification of the attack.

In the case of a real attack, a security analyst can use this GUI to find the packets which are indicative of an attack. The analyst could then read into why these packets are being classified as an attack and can use the provided information to validate whether the model's classification makes sense or not. Based on the determined type of attack, the analyst can then take the appropriate steps to stop this attack from continuing. The difference between my model and that of softwares like CrowdStrike is the transparency of the model's decision making process. Rather than having to fully trust that the model is accurate, the security analyst can use the present explanations to determine what attack is really occurring, and therefore take the correct steps to address the real ongoing attack.

The explainable features of this software can also be useful for improving the tuning of the model. Developers can find cases of inaccuracies in the model's predictions and use the explanations to figure out which features are responsible for reducing the model's accuracy. This information can then be used to alter the model's features and their respective weights.

In the future, this project could be expanded to include options for filtering data so that security analysts can quickly find patterns in the results. Filtering results would significantly speed up the process of finding attacks and comparing network packets against one another.

Finally, it is important to address that this project is not intended for use in the real world, but is rather an exploration of how NIDS systems that incorporate machine learning could be made to provide more transparency. Clearly, this project's scope is limited to only a portion of DDoS attacks, and would therefore not be very useful to a real organization that is facing countless varieties of attacks, however, my project shows that it is possible and feasible to display the results of an ML model's inner mechanisms. Larger, more well-implemented IDS softwares that employ ML models can use this research to enhance their explanations of their models' classifications and rationale. This implementation would not only improve users' trust of the ML aspect of the software, but also assist users in their investigation of anomalous packets.

Sources

GfG. “Intrusion Detection System Using Machine Learning Algorithms.” *GeeksforGeeks*, GeeksforGeeks, 14 Jan. 2022, www.geeksforgeeks.org/intrusion-detection-system-using-machine-learning-algorithm.

GfG. “Stacking in Machine Learning.” GeeksforGeeks, GeeksforGeeks, 20 May 2019, www.geeksforgeeks.org/stacking-in-machine-learning/.

Hindawi. “Table 3: Malware Detection in Self-Driving Vehicles Using Machine Learning Algorithms.” Table 3 | Malware Detection in Self-Driving Vehicles Using Machine Learning Algorithms, www.hindawi.com/journals/jat/2020/3035741/tab3/.

H. Holm, "Signature Based Intrusion Detection for Zero-Day Attacks: (Not) A Closed Chapter?," 2014 47th Hawaii International Conference on System Sciences, Waikoloa, HI, USA, 2014, pp. 4895-4904, doi: 10.1109/HICSS.2014.600.

Jacky. “What Is Network Intrusion Detection System (NIDS)?” Wordpress-331244-3913986.Cloudwaysapps.Com, 25 Oct. 2023, www.sapphire.net/cybersecurity/nids/#:~:text=NIDS%20works%20by%20examining%20data,infections%2C%20and%20unauthorized%20access%20attempts.

Jallad, Khloud Al, et al. “Anomaly Detection Optimization Using Big Data and Deep Learning to Reduce False-Positive - Journal of Big Data.” SpringerOpen, Springer International Publishing, 31 Aug. 2020, journalofbigdata.springeropen.com/articles/10.1186/s40537-020-00346-1.

Neupane, Subash, et al. “Explainable Intrusion Detection Systems (X-IDS): A Survey of Current Methods, Challenges, and Opportunities.” arXiv.Org, 13 July 2022, arxiv.org/abs/2207.06236.

Othman, Suad Mohammed, et al. “Intrusion Detection Model Using Machine Learning Algorithm on Big Data Environment - Journal of Big Data.” *SpringerOpen*, Springer International Publishing, 24 Sept. 2018, journalofbigdata.springeropen.com/articles/10.1186/s40537-018-0145-4.

S, Lavanya. “Gaussian Naive Bayes Algorithm for Credit Risk Modelling.” Analytics Vidhya, 1 Mar. 2022, www.analyticsvidhya.com/blog/2022/03/gaussian-naive-bayes-algorithm-for-credit-risk-modelling/#:~:text=It%20is%20a%20supervised%20machine,it%20is%20a%20probabilistic%20classifier.

Yang, Yi, and Abdallah Shami. “IDS-ML: An Open Source Code for Intrusion Detection System Development Using Machine Learning.” *Software Impacts*, Elsevier, 22 Nov. 2022, www.sciencedirect.com/science/article/pii/S2665963822001300.

“How to Train a Learning Model.” *Pluralsight*, 16 Oct. 2019, www.pluralsight.com/blog/machine-learning/3-steps-train-machine-learning.

“Random Forest: A Complete Guide for Machine Learning.” Built In, builtin.com/data-science/random-forest-algorithm.

“Search UNB.” *University of New Brunswick Est.1785*, www.unb.ca/cic/datasets/ddos-2019.html.

“SYN Flood Ddos Attack.” Cloudflare, www.cloudflare.com/learning/ddos/syn-flood-ddos-attack/.

“UDP Flood Ddos Attack.” Cloudflare, www.cloudflare.com/learning/ddos/udp-flood-ddos-attack/.

“Understanding TCP Flags.” Site24x7, www.site24x7.com/learn/linux/tcp-flags.html#:~:text=connection%20is%20established-,The%20ACK%20flag,the%20number%20of%20bytes%20received.

“Understanding the 5 Types of Intrusion Detection Systems.” *Helixstorm*, 18 Aug. 2022, www.helixstorm.com/blog/types-of-intrusion-detection-systems/.

“Welcome to Abertay University’s Research Portal.” Abertay University, rke.abertay.ac.uk/.

“What Is a Distributed Denial-of-Service (Ddos) Attack?.” Cloudflare, www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

“What Is a Good Accuracy Score in Machine Learning?” Deepchecks, 13 Nov. 2022, deepchecks.com/question/what-is-a-good-accuracy-score-in-machine-learning/#:~:text=Industry%20standards%20are%20between%2070,various%20businesses%20and%20sectors’%20needs.

“What Is Explainable AI (XAI)?” IBM, www.ibm.com/topics/explainable-ai.