CIC-IDS-2017 Dataset Analysis

CT6045: Big Data Analytics

Josh Demir

A report submitted in partial fulfillment of the requirements for the degree of Business Computing at the University of Gloucestershire. It is substantially the result of my own work except where explicitly indicated in the text.

Contents

1 Introduction

1.1 Preamble

This report presents a supplementary discussion of a real-time classification model for the CIC-IDS-2017 dataset. This report contains discussion of the outputs of the code developed for the CT6045 assignment at the University of Gloucestershire.

All of the supporting code for this report is available on GitHub at `https://github.com/moosejaw/ct6045-assignment`. Although the code is included with this report in the assignment submission, I recommend that you use GitHub to clone the repository onto your system if you choose to follow along with this report by running the code yourself. The system specifications when running the code developed in this report are included in Appendix A.

All instructions regarding how to run the code effectively are included in the README.md file. It is highly recommended you read it in order to set your system up correctly to run the code.

1.2 Overview

The aim of this task is to create an effective statistical model for labelling incoming network packets as 'benign' or 'malicious.' The model should classify packets representing normal traffic (such as regular end-users or communication between devices on the network) into the benign class, and packets relating to malicious intent – such as denial-of-service (DDoS) or slowloris attacks – into the malicious class.

The dataset used as a basis for this task is the Intrusion Detection Evaluation dataset from *Sharafaldin et al.* (2018). Specifically, the 'GeneratedLabelledFlows' section is used, generated from processing outputs from `tcpdump` which have been converted using `CICFlowMeter` into `.csv` files. The dataset was captured over 5 days in July 2017, monitoring a typical network and the traffic sent through it. The dataset consists of mostly benign packets, although many malicious packets were observed, and these packets were classified according to the corresponding type of

attack. For example, a packet from a DoS attack is classifed as '`DoS Hulk`.'

This report describes the key charactersitics of the data, comparing and contrasting the typical attributes of each class. This report also contains more in-depth descriptive analysis of the dataset, including an analysis of the most significant IP addresses – demonstrating some of the key areas that analysts would want to observe in practice. To do so, this report contains findings from several different methods of analysis, including graph analysis and PageRank.

Included in this report is the design and development of a predictive model, experimentation with different algorithms and the results and implications of each. Additionally, a real-time predictive model and input pipeline is created to conduct analysis as and when packets arrive. Finally, the report concludes with a critical review of the tools and techniques used, plus limitations of this research and recommendations for further development.

It is important to note that throughout this report, we strive to develop a model which can perform binary classification on incoming data. In this context, identifying the specific types of malicious network attacks are irrelevant, and we aim to simply identify packets under two categories: benign or malicious.

2 Descriptive Analytics

This section seeks to provide descriptive analytics on the dataset. This is an important step in developing a predictive model, as it allows for a deeper understanding of the data's properties, including how we can extract the most meaningful information and apply it to our own purpose.

For this section, we will discuss the basic properties of the data and investigate ways in which we can extract key information, such as the IPs responsible for sending malicious packets and what their specific targets are inside our network. We then evaluate each feature's usefulness in a predictive model by performing inter-feature correlation. From this, we extract the features with the highest correlation scores and explain their implications for developing a predictive model.

2.1 Pre-processing

Related terminal commands:

```
./preprocessing.py
```

We begin by transforming the dataset to make it easier to work with down the line. First, we transform the headers to become more machine-readable: leading/trailing whitespace is removed, intermediate whitespace is replaced with underscore characters and the text is converted to lower case. The *flow ID* and *timestamp* fields are removed entirely.

We also cast each record to the floating point datatype. The exceptions are the *source IP*, *source port*, *destination IP* and *destination port* fields, as they contain descriptive data which cannot be converted to floats. It could be argued that the port fields could be converted to floating points, but this would be unnecessary as they are naturally discrete numbers and no arithmetic operations need to be applied on the numbers themselves.

The most important transformation is the 'label' field. In the initial dataset, the rows are classified as either benign or the type of network attack they were a part

```
raw_d = pd.read_csv('raw.csv')
raw_d.head()
```

	Flow ID	Source IP	Source Port	Destination IP	Destination Port	Protocol	Timestamp	Flow Duration	Total Fwd Packets	Ba P
0	192.168.10.12-192.168.10.50-35396-22-6	192.168.10.12	35396	192.168.10.50	22	6	7/7/2017 1:00	1266342	41	
1	192.168.10.16-192.168.10.50-60058-22-6	192.168.10.16	60058	192.168.10.50	22	6	7/7/2017 1:00	1319353	41	
2	192.168.10.12-192.168.10.50-35396-22-6	192.168.10.12	35396	192.168.10.50	22	6	7/7/2017 1:00	160	1	
3	192.168.10.12-192.168.10.50-35398-22-6	192.168.10.12	35398	192.168.10.50	22	6	7/7/2017 1:00	1303488	41	

4 rows × 85 columns

```
[3] pro_d = pd.read_csv('processed.csv')
    pro_d.head()
```

	source_ip	source_port	destination_ip	destination_port	flow_duration	total_fwd_packets	total_backward_packets	tot
0	192.168.10.12	35396	192.168.10.50	22	1266342.0	41.0	44.0	
1	192.168.10.16	60058	192.168.10.50	22	1319353.0	41.0	44.0	
2	192.168.10.12	35396	192.168.10.50	22	160.0	1.0	1.0	
3	192.168.10.12	35398	192.168.10.50	22	1303488.0	41.0	42.0	

Figure 1: Head prints of the raw and processed .csv files when loaded into Pandas dataframes.

of, such as DoS Hulk or Slowloris, implying a multi-class classification system. To reduce this to a binary classification, the labels are converted to floating point numbers: 0.0 and 1.0 for benign and 'malicious' packets respectively. These values for labels were chosen as 0.0 implies a negative classification and malicious implies a positive classification – and we are seeking to positively identify malicious packets in the model.

Figure 1 demonstrates the difference between the two files before and after the pre-processing is performed. The top print shows the initial dataset, with capitalised header fields, spaces between each word, and integers for the numerical fields. To that end, the bottom print shows the dataset after it has been processed: the header names are subsequently easier to work with and the numerical data has been converted to floating point numbers. The irrelevant fields have also been removed.

2.2 Determining the most significant addresses

Related terminal commands:

```
./pagerank.py
```

Now that the dataset has been transformed, we can begin to investigate it for practical purposes. The first is to identify the most significant IP addresses within the dataset. 'Significant' refers to both the internal and external IP addresses in our network. The most significant source IP addresses would refer to those likely responsible for sending many packets to our network, indicatingvarious types of DoS attacks, of which there are many different kinds (*Dileep*, 2016). The most significant destination IP addresses would represent the individual targets of network attacks.

For our approach, 'significance' is measured based on the frequency in which an address appears within samples we take from the data. As discussed before, this metric is an easy way to identify malicious sources and potentially vulnerable destinations.

To measure this, we use an approach derived from PageRank in order to determine the probabilistic eigenvector of each unique route (source IP $\rightarrow$ destination IP) that each address has taken, for both the source and destination IP columns. We begin by taking k number of random samples of size x from the dataset.

We then extract each unique address from each address column to form a set of size n, counting the number of times each has sent/received a packet and storing it as a collection of key-value pairs. For each iteration, an eigenvalue is determined by $1/kx$. This eigenvalue is multiplied across each address count in the collection to form a probabilistic eigenvector E where:

$$\sum_{i=1}^{n} E_1 + ... + E_n = 1$$

This principle is demonstrated in figure 2 where the number of observations of a particular dummy IP address are converted to probabilities.

IP address	Frequency	Weighted eigenvector
10.0.2.17	13	~ 0.38
12.7.24.24	6	~ 0.18
19.21.5.2	2	~ 0.06
10.0.5.40	10	~ 0.30
32.18.9.2	2	~ 0.06
10.7.2.254	1	~ 0.03
Total frequency	34	

Figure 2: Table showing the frequency-to-eigenvector conversion on a set of dummy data.

For each sample, the eigenvector for each IP address is recorded. Then, each set is concatenated into one. In cases where an IP address appeared in more than one sample, its average value is recorded. In the code, the top 10 addresses with the highest value are displayed. For the source IP analysis, IP addresses beginning with 192.168 were excluded as we wanted to keep focus on IP addresses external to the network.

2.2.1 Results and interpretation

When running the code, it returns the results shown in figure 3.

Starting with the source IP column results, we can see that there is a distinct IP address amongst the samples collected: 172.16.0.1, with an average probability score of 0.25. In one sample, this score would attribute over a quarter of the total probability – a considerable amount – but more damning are scores of the other 'top' IP addresses which are significantly lower. From this, we can conclude that 172.16.0.1 has sent an extremely large number of packets over the network relative to the other addresses collected in the samples. It is likely that this address was issuing a DoS attack.

Moving onto destination IP, there are two key players: 192.168.10.3 and 192.168.10.50. Their scores are similar to that of 172.16.0.1, however the explanations behind this high amount of traffic are more reasonable. First, it is likely that either address represents the web endpoints (where most users would

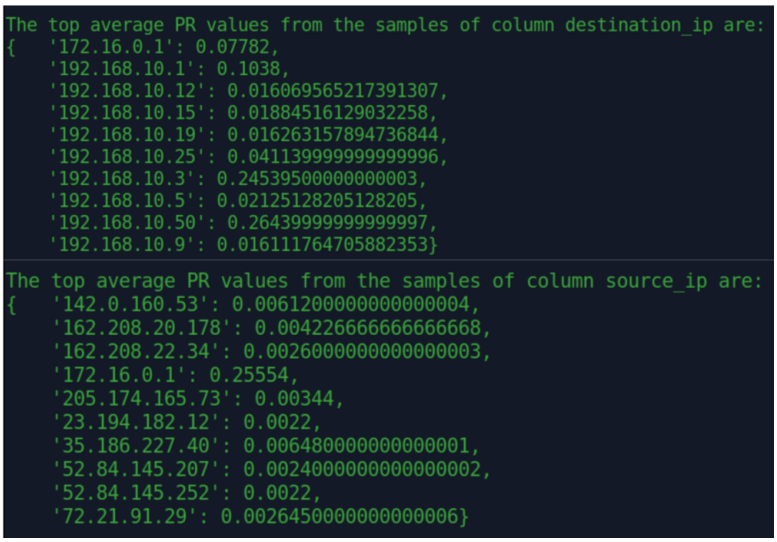

Figure 3: Terminal output from pagerank.py.

access the network) or a service where there are a high amount of transactions (such as a database). It is equally likely that these addresses could be victims of a DDoS attack. Other points of interest are 192.168.0.1 with a score of 0.1, indicating this address too saw a significant amount of traffic. 172.16.0.1 also appears in this list, indicating the network was sending many communications back to the address. This further suggests a brute-force DDoS attack took place over something like a Slowloris attack where packets are held open for a significant amount of time.

This approach is reasonable to discover some significant IP addresses, although equally as effective could be gathering simple statistics such as modal values for different samples and simply taking the frequency at which each address has sent/received packets. However, converting this value to a weight is useful to see the proportion of the sample the address represented. Taking an average score for every address across each sample was also ineffective as the random nature of the samples meant that a significant IP address could easily be missed if it appeared in less samples than another. Despite this, the sheer weight of addresses such as 172.16.0.1 compared to others are still heavily implicit of a brute-force attack.

```
The most common routes for IP 172.16.0.1 (as a source_ip) are:
{   '/172.16.0.1/192.168.10.15': 5,
    '/172.16.0.1/192.168.10.50': 38143,
    '/172.16.0.1/192.168.10.51': 168,
    '/172.16.0.1/192.168.10.8': 6,
    '/172.16.0.1/192.168.10.9': 5}
```

Figure 4: Terminal output containing IP address routes and their number of observations.

2.3 Graph analysis

Related terminal commands:

```
spark-submit --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11 graph_analysis.py
```

Another way of measuring some of the key traffic throughout the network is to perform a graph analysis. With this approach, we can take a more in-depth look at the typical routes of the significant IP addresses found in the previous section. We begin by re-using the same samples from the previous section - they can be re-created in the new script by sampling the data with the same random seeds. Then, we load in each 'significant IP' from the previous section into an array corresponding to whether it was a source or destination IP. We again count the unique routes of each IP and display the top 5 for each, as shown in figure 4. In this case, the key format represents $/SourceIP/Dest.IP/$ and the value is the number of observations.

We can then use these observations to build the graphs. We build one graph each based on source and destination IP address respectively. Each unique IP address that appears throughout the observations is assigned a vertex in the graph. We can then use the associated IDs of the vertices to create the edges, representing communication from one IP address to another. An associated weight is given formed of a string representing 'high', 'medium' , or 'low' based on the number of observations of the route.

Once each graph is created, we can observe the degrees of each vertex. The degrees represent the total number of inputs and outputs for each IP address, thereby

showing us the addresses which saw the most traffic overall. PageRank is also available. While it still offers some level of insight into high-traffic addresses, it is not a particularly effective metric in this instance as we have only produced one edge for every unique route observation, and we cannot assign our own numerical weights to the graph, so the significance of many important IP addresses is lost, shown in the difference between IDs 1 and 6 in the PageRank column of figure 5. We know that from the previous section, 172.16.0.1 is the most suspect IP address, but 192.168.10.9 was a diverse target for traffic, so it therefore received the highest PageRank score.

Note that for each graph, all outputs are truncated to the top 20 rows.

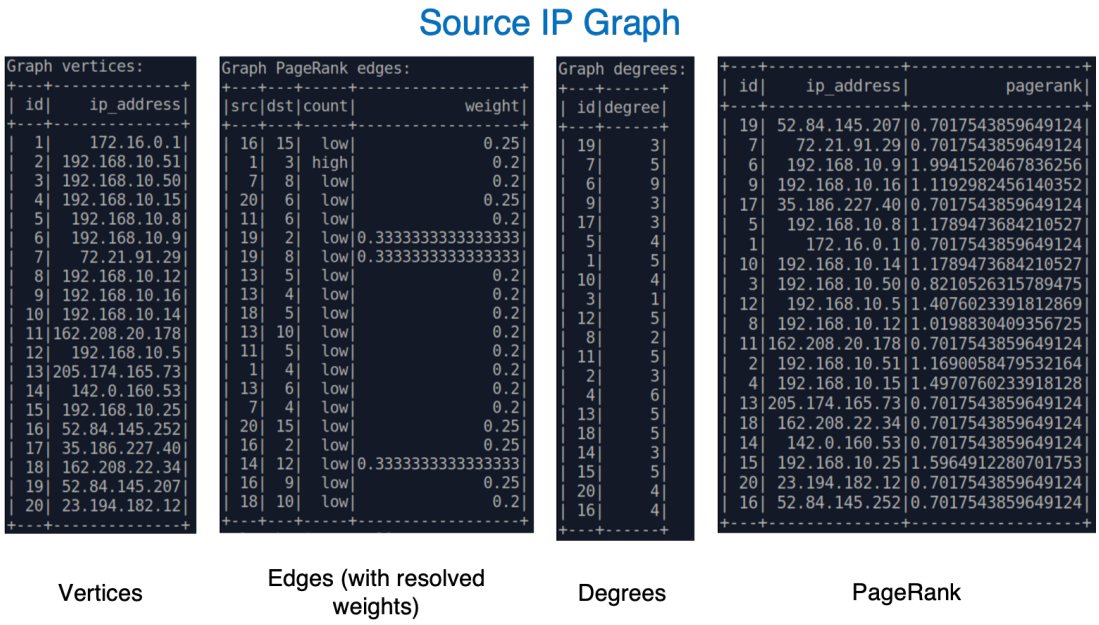

Figure 5: Source IP graph output.

2.3.1 Results and interpretation

These particular graphs help us to further investigate some of the key facts we already know. By evaluating our samples in the previous section we were able to determine the most suspect and most vulnerable machines. By building the graph, we can extract some further useful information including the total number of inputs

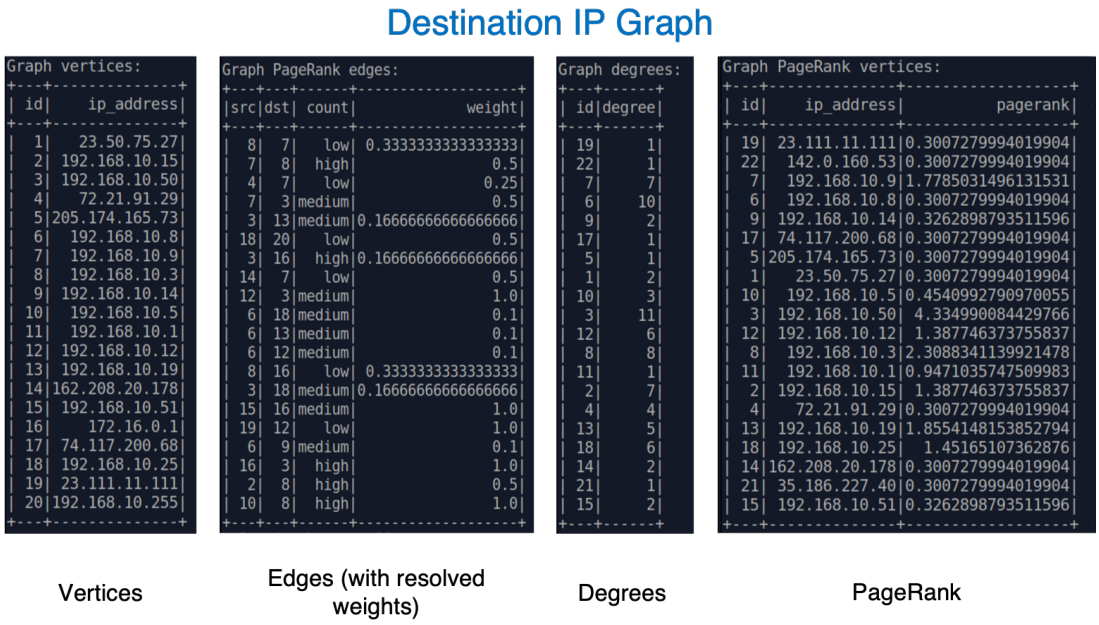

Figure 6: Destination IP graph output.

and outputs for each address, indicated by the number of degrees. However, from the unidirectional nature of the graph, it is likely that the nodes with the largest degrees would be those internal to the network (which is true). This is because each instance of the address receiving traffic is recorded, whereas source IPs are bound to a degree limit of 5, as we only chose to observe the top 5 routes of each source IP. This makes the analysis useful for observing how packets move around the network, particularly when the internal network nodes receive traffic, but we have limited insight into external traffic coming into the network.

To an extent, this makes PageRank useful only for destination IP addresses in both graphs. In figure 5, 172.16.0.1 has the same PageRank as the other external addresses. However, the influence of internal nodes is clear in both graphs. In figure 5, the highest PageRank was 192.168.10.9, indicating this saw a lot of traffic throughout the observations and is a potentially vulnerable target. However, the PageRank scores in 6 – where we are truly attempting to observe vulnerable targets – are significantly larger than its score of 1.99. 192.168.10.50 had the largest score with 4.33, and 192.168.10.3 was second with 2.30. The difference between

these two scores is significant still, indicating a much more diverse number of connections to 192.168.10.50. However, while these findings appear intriguing at first glance, some further investigation may likely reveal a banal truth such as 192.68.10.50 being the web front-end of the network or some other internet-facing machine. We could investigate this likelihood by simply observing the port numbers, as 80 or 443 would be clear indicators of a web server. Perhaps it is these intermediate PageRank values, such as 192.168.10.3 which are hosting vulnerable services which attackers are seeking to exploit, but further investigation of the data would be required to provide answers. Going forward, we could further refine these graphs by concatenating destination IP address and ports to reveal truly vulnerable destinations, and it would be beneficial to be able to assign our own numerical (and not categorical) weights to the graph from the beginning, so we can make PageRank more useful across the board.

2.4 Inter-feature correlation

Related terminal commands:

```
./feature_reduction.py
```

As the processed dataset consists of 51 numerical columns in total, it would be appropriate to reduce the number of features we send to a predictive model. There are various methods of reducing dimensionality of the data, but one which could provide some useful insight into relationships between fields is *inter-feature correlation*. We begin by taking samples of size x from each file in the dataset – of which there are 8. Each sample is concatenated in order to form a dataframe of dimensions $51 \times 8x$.

From the newly-formed dataframe, we run the built-in correlation function available in the `pandas` library, outputting a correlation matrix in dataframe format. The correlation scores are appended to an array and this process is repeated k times.

For each matrix in the array, the mean value of each cell is calculated, thus representing the mean inter-feature correlation scores from the samples collected.

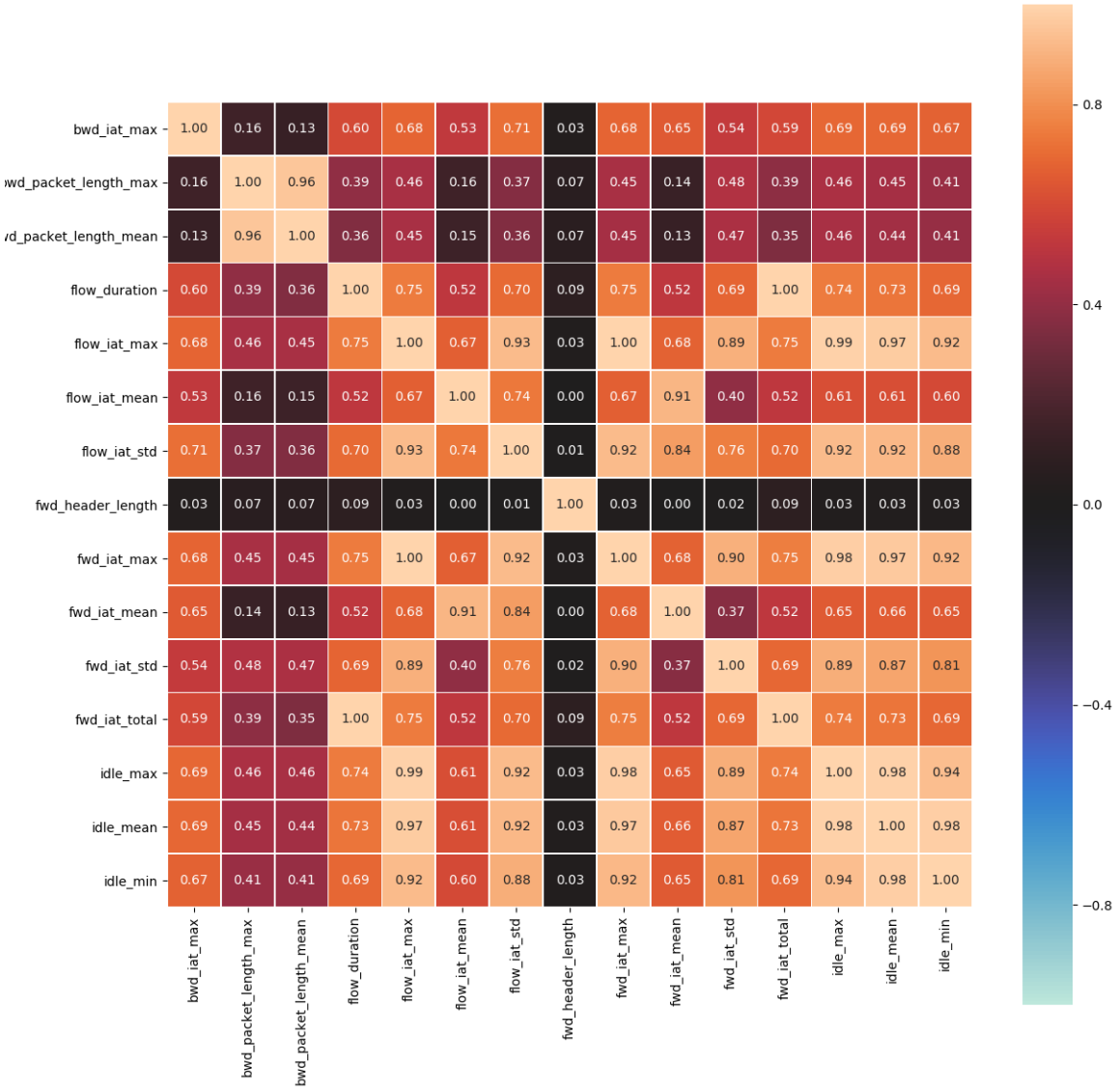

Figure 7: Correlation matrix containing the top 15 features with the highest counts of high correlation.

The dataframe is then sorted by headers in alphabetical order to create a symmetrical matrix for plotting.

A subsequent matrix plot is created using `matplotlib` and the `seaborn` library which provides convenient wrapper functions for plotting matrices and heatmaps. We then refine this matrix to reflect only the features with the largest number of

highly-correlated features. For each column in the correlation matrix, a count is recorded for each observation of a correlation score of $< |0.6|$, indicating a highly inter-correlated feature. From the results of this function, we extract the top 15 features with the highest counts and plot them into a separate heatmap, shown in figure 7.

If the output matrix shows a reasonable number of high correlation coefficients between the selected features, then we can use these features as the basis for forming our prediction model.

2.4.1 Results and interpretation

Fortunately, the output correlation matrix shows generally strong inter-feature correlation between the top 15 features. The one obvious exception is `fwd\_header\_length` which must have been a highly-correlated feature to appear on the matrix, however it does not have any significant correlation with the other features included here. Therefore, it would be wise to remove this column from the prediction model going forward, thereby reducing our number of input dimensions to 14.

Some other general observations are that every correlation coefficient is positive, which indicates that each feature rises and falls in proportion to others. This makes logical sense, as features like maximum packet lengths and mean packet lengths would naturally be proportional – as a maximum value increases, it raises the mean with it. Particularly high correlation scores are observed in `idle\_` and `flow\_iat` features, which contain some scores significantly close to 1, indicating very strong relationships.

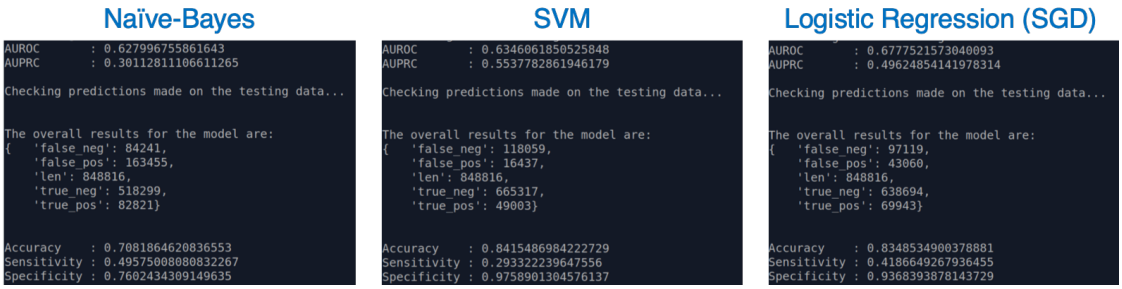

Figure 8: Comparison of standard performance metrics of Naive-Bayes, SVM and Logistic Regression.

3 Predictive Analytics

3.1 Testing different algorithms

Related terminal commands:

```
./train_test_split.py
spark-submit prediction_model.py
```

We begin by testing different algorithms with our current specification: we first split the dataset into a training and testing set, weighted at 70% and 30% respectively. Then, we use Apache Spark with standard `SparkContext` to declare each model – Naive-Bayes, SVM, and Logistic Regression with SGD. For the initial testing phase, the hyperparameters of each model are kept at their default setting, according to their Spark implementation, except for the number of iterations when training which has been increased to 1000 to ensure a much better all-round fit of the dataset.

Each model maps training and testing data into `RDD`s. The elements of the training RDD are converted into `LabeledPoints` where the label column is separated from the rest of the features, which are converted into a vector. In the testing RDD, the output is mapped to a tuple of format `(prediction, true label)`, allowing for easier collection of model statistics. The outputs of each model are shown in figure 8.

3.1.1 Results and interpretation

The results of each test are somewhat promising for developing our own model. The worst performer was Naive-Bayes, which returned an accuracy of 70%, along with the smallest AUROC of 0.62. Despite its low overall accuracy, it outperformed the other two algorithms in sensitivity, suggesting the algorithm was better suited at detecting malicious packets in general (*Parikh et al.*, 2008). The number of false negative classifications were lower than the other two models, and the number of true positives detected was higher. However, a score of 0.49 for sensitivity is still indicative of an inefficient classification model, reflected in the low AUROC.

SVM returned the largest overall accuracy at 84%, but still presented some glaring flaws. SVM was especially good for specificity – a score of 0.97 is respectable, but conversely the sensitivity is too low to ignore. The high specificity tells us that SVM is particularly good at identifying benign packets but severely lacks in ability to detect malicious ones. We can see in the numbers that 49003 correct positive identifications vs. 118059 incorrect ones shows this clearly. Overall, the trade-off between low specificity and high sensitivity balances out as the AUROC is only larger than Naive-Bayes by 0.02. AUROC is one of the most significant metrics as it tells us the probability at which a randomly-selected positive packet would be classified as more likely to be malicious than a benign packet (*Hajian-Tilaki*, 2013), so we want this to be as close to 1 as possible.

The model returning the largest AUROC was logistic regression. This difference is still roughly 0.04, indicating it is only slightly more effective at classifying positive packets. The model represents more of a middle ground between the other two. Naive-Bayes was relatively effective at finding positives and SVM was effective at finding negatives, but the scores for sensitivity and specificity of logistic regression lie between the values for each of the other models. Logistic regression was still effective at finding negative packets with a specificity score of 0.93. Even still, these initial results seem promising. We could infer from the scores that logistic regression had been underfit, so by increasing the iterations of training plus tweaking some other parameters, we could more effectively optimise this model.

3.2 Streaming model

```
Refer to README.md for instructions on how to construct the
    pipeline.
```

For real-time streaming, Apache Spark provides some basic streaming models to work in conjunction with different file systems such as HDFS. In this situation, `StreamingLogisticRegressionWithSGD` is used.

The streaming model uses all numerical columns from the processed dataset instead of just those found with inter-feature correlation. The justification for doing so is explained in the conclusion.

The purpose of the streaming model is to analyse packets as and when they arrive. We first construct a virtual network in order to produce packets we can stream to the model. This is achieved with a Docker virtual network, where one container runs an Apache web server with a dummy index page, and we create other containers which routinely send packets to the web container. These containers are `normal_traffic` which periodically opens normal requests to the index page, `ddos` which periodically sends 1490 bytes to the target, and `slowloris` by *Yaltirakli* (2020) which issues a slowloris attack.

Docker constructs its own network interface for communication between packets, so we use `TCPDUMP and CICFlowMeter` by *Tiwatthanont* (2020) to listen on the interface and produce `.pcap` files which are automatically converted to `.csv` files analogous to those in the initial dataset – as they contain the same features and characteristics. The tool used for doing this is NetFlowMeter by *Lashkari et al.* (2020).

A Python script then routinely checks for new `.csv` files being produced and then copies the files over to the appropriate directory (training or testing) in HDFS. The streaming model has data streams listening on these directories in HDFS, so incoming data will be used for training or testing respectively. This is shown in

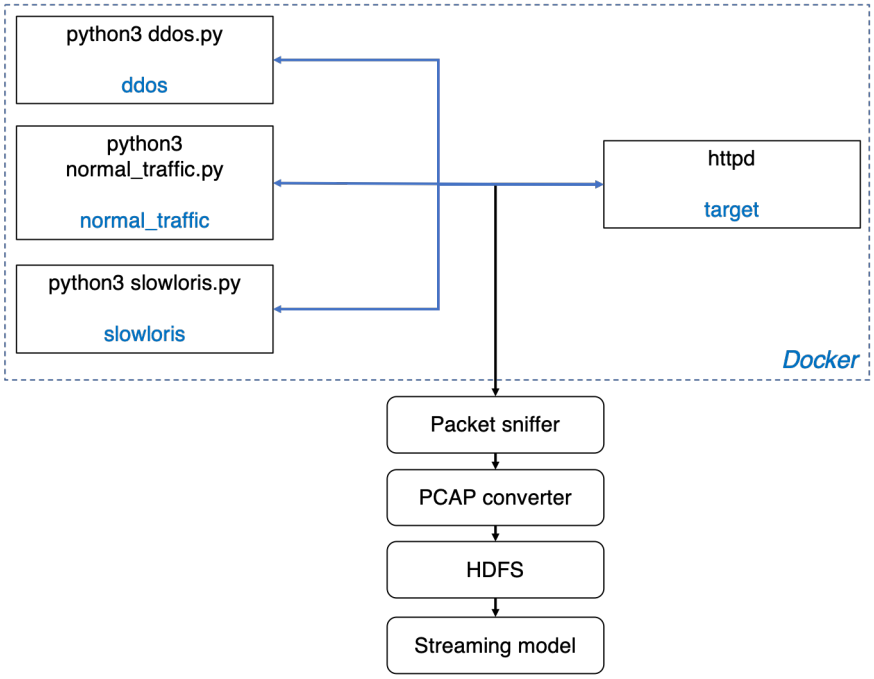

Figure 9: Pipeline for the streaming model.

figure 9.

3.2.1 Results and interpretation

The streaming model is slightly more difficult to work with in terms of statistics, as the only reliable output from the model is periodic prediction results, shown in figure 10 where the output is formatted as `(label, prediction)`. Spark's recommendation for sending predictions for processing is to open a connection and send the data that way. `rta_listener.py` sought to do this, but due to the nature of sockets closing when they do not receive consistent data, plus Python failing silently when this happens, there is a lack of synchronisation between the model and the listener, so it proved an inefficient way of gaining insight from the model. An example of calculated output from the untrained model is shown in figure 11.

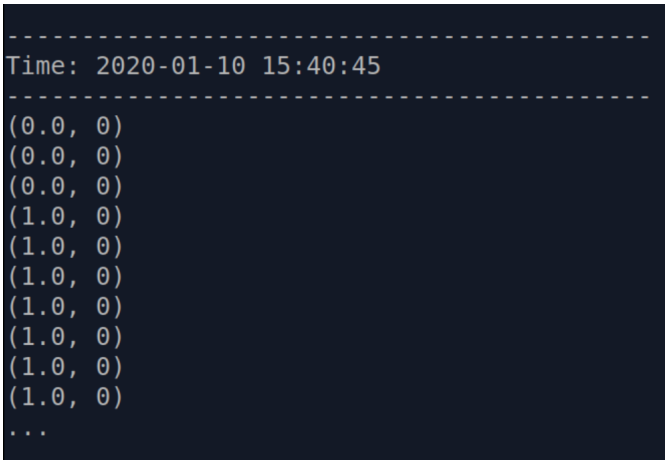

Figure 10: Printed output of the untrained model's prediction `dstreams`

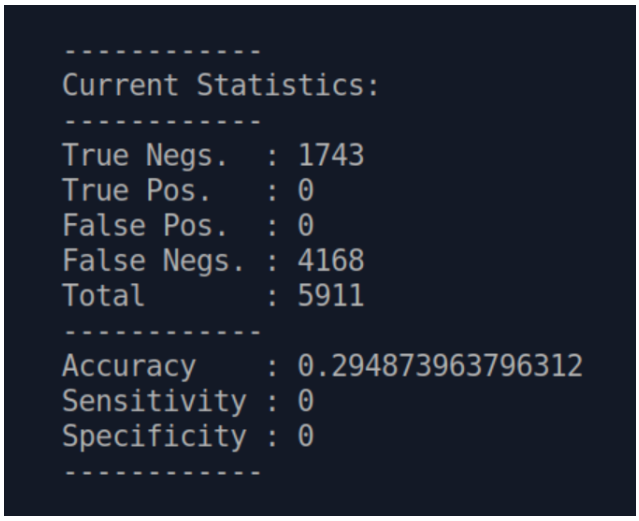

Figure 11: Output of the predictions vs. labels received from the untrained model.

These statistics prove ineffective when evaluating the ability of the streaming model. Therefore, a new script, `generated_prediction_model.py` was created along with custom training data to effectively evaluate the model, tested on a single set of input data. We can make the assumption that an appropriately trained streaming model would carry similar statistics. In the generated model, we use 5000 iterations (representing a high amount of training data) and enable the `intercept` parameter for the logistic regression model, which vastly improved the sensitivity. This is likely due to the imbalance of benign and malicious training data – as more benign packets were observed during training so the model was underfit when finding malicious packets. By including the intercept in the model, we see a vast improvement when detecting malicious packets. The intercept parameter allows particular features to have their own unique bias in the model, and that a prediction where all features = 0 is not necessarily 0 too. The output of the highly-optimised model is shown in figure 12.

```
Predicting on the testing data...
AUROC       : 0.957608695652174
AUPRC       : 0.9392523364485982

Checking predictions made on the testing data...


The overall results for the model are:
{'false_neg': 0, 'false_pos': 39, 'len': 1063, 'true_neg': 421, 'true_pos': 603}


Accuracy    : 0.9633113828786454
Sensitivity : 1.0
Specificity : 0.9152173913043479
```

Figure 12: Output of the generated model script showing the potential for the streaming model.

We opt to show the output of one batch of data for this model as it better reflects how we should measure the streaming model. The streaming model is able to train and predict in parallel, so if we were to evaluate the cumulative number of classifications the model has made, there would be a significantly high number of incorrect classifications to begin with. Although, as the model begins to train and make more correct predictions, this number would begin to balance out – however this would take a signficant amount of time.

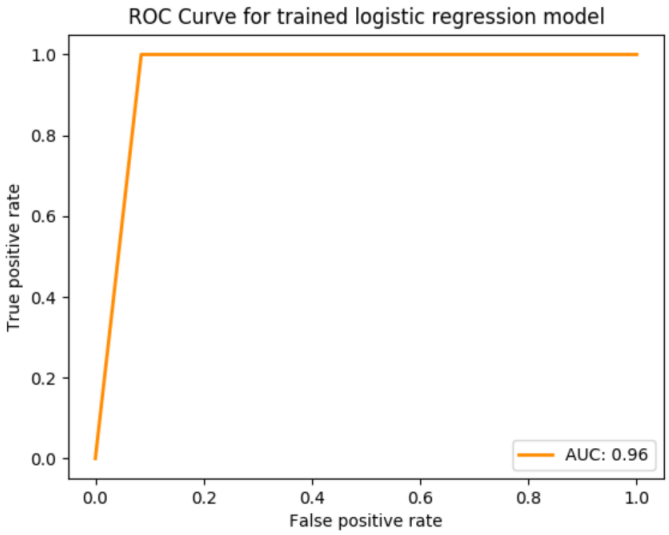

Figure 13: ROC curve for the optimised logistic regression model.

The results are very interesting. The senstivity has skyrocketed from around 0.4 in the experimentation phase to 1.0, meaning that model is now perfectly fitted toward identifying positive packets. However, the specificity has reduced slightly, suggesting that the inclusion of the intercept has had so much of a shift towards positive predictions, it has influenced the model negatively when trying to predict benign packets.

Overall, the 96% accuracy is a respectable number but more intriguing is the AUROC. Due to the senstivity being 1.0 we can observe a high-scoring ROC curve, shown in figure 13. The ROC curve is promising – an AUC of 1.0 would represent a perfectly-fitted model. The true positive rate provides most of the area of the curve by being mostly straight. However, the pitfall of the model lies in the false positive rate, meaning that it is somewhat likely that randomly-selected negative packets could be classified as false positives. For our particular scenario, we could argue that a model which efficiently detects malicious packets is more valuable – which it certianly achieves through its high sensitivity score. In practice, should this model return many false positives, we could further investigate if there is a cause for alarm by further analysing the behaviors of the IP address in question, using techniques discussed in the descriptive analytics section.

4 Conclusion and Recommendations

This report reflects many of the topics discussed in CT6045, particularly when seeking to create a real-time predictive model. Apache Spark with Hadoop proved a useful toolkit for performing this, although some critical issues arose. For the streaming model, the class required does not support the `intercept` parameter, meaning that we would be unable to activate feature bias, limiting us to a poorly-performing real-time model. This is why in section 3.2.1, the results of a pre-generated model were shown – to demonstrate the potential of the streaming model with some tweaks, but the model is ineffective from the start when Spark does not yet support such a critical parameter. It should be noted that other parameters for logistic regression – such as experimenting with regularisation and step size did not produce any significant effect on the results. In fact, performance suffered by changing regularisation from L2 to L1, indicating that these parameters are already optimised when at their default settings in Spark.

Ultimately, all of the numerical dataset fields were used in the prediction model as the higher accuracy, sensitivity and specificity scores were observed after doing so. This indicates that perhaps inter-feature correlation was not the best way to determine the input features to use in the model, and that a process such as Principal Component Analysis may have produced better results in this case.

The data pipeline is not truly 'real-time' either. A lot of pre-processing must be performed after capturing packets. A long latency from packet observation to being processed by the model exists as the packet must first be observed, then written to a `.pcap` file, then converted to a `.csv` file, and then copied manually to the appropriate directory using the `hdfs dfs` utility. It would be much more beneficial in practice to use an optimised pipeline, such as the one proposed by *Zhang et al.* (2014), where they found increased performance compared to standard HDFS as it creates multiple pipelines for each incoming data block. Some more obvious changes would be to use dedicated Spark and Hadoop clusters to further boost performance of write operations.

It would also be more beneficial to observe a real network when developing a

classification model. By creating our own virtual network, it is easy to homogenise the packets being sent across it. In essence, it is much easier for our model to fit to our data because the different classes will often share very similar data, whereas a true-to-life network in practice would see a much more diverse range of data coming in and out.

The descriptive analytics methods were useful for providing unique glances of data we have already collected. Forming probabilistic eigenvectors of packet routes is a good way of capturing 'significance' in addresses, which is useful in practice as it would allow users to specifically identify suspicious IP addresses based on a more meaningful metric than frequency – as probabilities show the proportion of a dataset in which the address was observed. PageRank itself was only so useful in this scenario for two main reasons: the first was that the graph was unidirectional, so PageRank immediately becomes less insightful as it also counts incoming edges for each node within a graph (*Page et al.*, 1999), and second was that it did not factor in weighted edges as we intended. This burden does also fall onto the way in which the graph was implemented. To create the 'significance' factors ourselves, we could implement a multigraph where each node has one outgoing edge to another node based on how many times the route was observed. In our case, say address 10.0.0.1 connected to 192.168.0.25 50 times – we would create the same edge 50 times. Of course, the clear disadvantage here is that it would exponentially increase the size of our graph and make computation functions (like PageRank) much more expensive.

Overall, by refining and condensing some of the practices observed throughout this project, an effective real-time model for detecting network attacks could reasonably be produced. However, the burden of choosing which elements to analyse in more detail and those not – from the perspective of both the incoming data and the hyperparameters of the model – would largely fall onto the developer.

References

Dileep, K. G. (2016), *Network Security Attacks and Countermeasures*, chap. DDoS Attacks and Their Types, pp. 197–206, eBook: IGI Global.

Hajian-Tilaki, K. (2013), Receiver operating characteristic (roc) curve analysis for medical diagnostic test evaluation, *Caspian Journal of Internal Medicine*, *4*, 627–635.

Lashkari, A. H., D. G. Gil, M. S. I. Mamun, Y. Zang, and G. Owhuo (2020), Netflowmeter, Online, Available at: http://netflowmeter.ca/.

Page, L., S. Brin, R. Motwani, and T. Winograd (1999), The pagerank citation ranking: Bringing order to the web., *Tech. rep.*, Stanford InfoLab.

Parikh, R., A. Mathai, S. Parikh, G. C. Sekhar, and R. Thomas (2008), Understanding and using sensitivity, specificity and predictive values, *Indian Journal of Opthalmology*, *56*, 45–50, doi:10.4103/0301-4738.37595.

Sharafaldin, I., A. Habibi Lashkari, and A. A. Ghorbani (2018), Toward generating a new intrusion detection dataset and intrusion traffic characterization, in *4th International Conference on Information Systems Security and Privacy (ICISSP)*, Portugal.

Tiwatthanont, P. (2020), Tcpdump and cicflowmeter, Online, Available at: https://github.com/iPAS/TCPDUMP_and_CICFlowMeter.

Yaltirakli, G. (2020), slowloris, Online, Available at: https://github.com/gkbrk/slowloris.

Zhang, H., L. Wang, and H. Huang (2014), Smarth: Enabling multi-pipeline data transfer in hdfs, in *2014 43rd International Conference on Parallel Processing*, pp. 30–39, IEEE.

Appendices

A Machine Specification

The implementation of the model was performed inside a virtual machine running on a macOS Catalina host. The specification of the system is listed below:

Virtual machine configuration

- **OS:** Ubuntu 18.04
- **RAM:** 4 GB Virtual / 8 GB Host
- **Processor:** 2 cores enabled Dual-Core Intel Core M 1.1GHz (Host machine)
- **Python version:** 3.6
- **Command line:** `bash`

Hadoop and Spark configuration

Apache Hadoop and Spark were installed onto the virtual machine only. As such, they only ran on `localhost` and no other machines.

- **Hadoop version:** 3.1.2
- **Spark version:** 2.4.4 (pre-built for Hadoop 2.7)