UNIVERSITY OF PISA

Artificial intelligence and data engineering

Data mining and machine-learning project

Intrusion detection system using Machine-Learning techniques to find malicious requests

Authors :

Alessandro Versari
Francesco Londretti

Year 2022/2023

Contents

Chapter 1

Introduction

Since internet existed, security has always been a concern.

As technology advances, so do the methods used by attackers to harm those who communicate online. Personal and sensitive information is at risk.

To protect users, security measures have been implemented, yet they are not always sufficient to prevent attacks. This project aims to develop two classifiers: one that can detect suspicious server requests (binary classifier) and the other that can identify the category of an attack (multi-class classifier).

These classifiers could be used in a proxy deployed at the edge of a system infrastructure. And used in collaboration with other cyber-security measures.

To train these models we found a data-set that contains some common attack classes.

Then we tested these models in a real world scenario that is a "CTF(Capture the flag) attack and defence".

By doing that we checked if our study could be used in a more generic situation or if the models that we produced could be only used in a "lab like" scenario.

We encountered multiple challenges while implementing our program for practical use in real-world systems, which will be discussed in further detail later in the report.

Chapter 2

Data overview

The data-set used for the training is the $UNSW - NB15$ data-set. It was created by applying *IXIA PerfectStorm* tool on a network. It includes nine categories of modern attack types and uses a realistic representation of normal traffic.

The data-set is composed by 2.540.047 entries and 47 features, as well as two types of label: "**attack_cat**" which represents the category of the attack and "**Label**" which represents if a request is an attack or not.

2.1 Feature description

The features are divided in 5 classes: Flow, Basic, Content, Time, Additional Generated.

In the next pages there will be some tables to describe them more.

2.1.1 Flow features

#	**Name**	**Type**	**Description**
1	*srcip*	nominal	Source IP address
2	*sport*	integer	Source port number
3	*dstip*	nominal	Destination IP address
4	*dsport*	integer	Destination port number
5	*proto*	nominal	Transaction protocol

2.1.2 Basic features

#	**Name**	**Type**	**Description**
6	*state*	nominal	Indicates to the state and its dependent protocol
7	*dur*	Float	Record total duration
8	*sbytes*	Integer	Source to destination transaction bytes
9	*dbytes*	Integer	Destination to source transaction bytes
10	*sttl*	Integer	Source to destination time to live value
11	*dttl*	Integer	Destination to source time to live value
12	*sloss*	Integer	Source packets re-transmitted or dropped
13	*dloss*	Integer	Destination packets re-transmitted or dropped
14	*service*	nominal	$http$, ftp, $smtp$, ssh, dns, $ftp-data$, ir and $-$ if not used
15	*Sload*	Float	Source bits per second
16	*Dload*	Float	Destination bits per second
17	*Spkts*	integer	Source to destination packet count
18	*Dpkts*	integer	Destination to source packet count

2.1.3 Content features

#	Name	Type	Description
19	*swin*	integer	Source TCP window advertisement value
20	*dwin*	integer	Destination TCP window advertisement value
21	*stcpb*	integer	Source TCP base sequence number
22	*dtcpb*	integer	Destination TCP base sequence number
23	*smeansz*	integer	Mean of the flow packet size transmitted by the *src*
24	*dmeansz*	integer	Mean of the flow packet size transmitted by the *dst*
25	*trans_depth*	integer	Represents the pipelined depth into the connection of http request/response transaction
26	*res_bdy_len*	integer	Actual uncompressed content size of the data transferred from the servers http service.

2.1.4 Time features

#	Name	Type	Description
27	*Sjit*	Float	Source jitter (mSec)
28	*Djit*	Float	Destination jitter (mSec)
29	*Stime*	Timestamp	record start time
30	*Ltime*	Timestamp	record last time
31	*Sintpkt*	Float	Source interpacket arrival time (mSec)
32	*Dintpkt*	Float	Destination interpacket arrival time (mSec)
33	*tcprtt*	Float	TCP connection setup round-trip time, the sum of SYN and $ACKDAT$
34	*synack*	Float	TCP connection setup time, the time between the SYN and the SYN_ACK packets.
35	*ackdat*	Float	TCP connection setup time, the time between the SYN_ACK and the ACK packets.

2.1.5 Additional/Generated features

#	Name	Type	Description
36	*is_sm_ips_ports*	Binary	If source (1) and destination (3) IP addresses equal and port numbers (2)(4) equal then 1 else 0
37	*ct_state_ttl*	Integer	No. for each state (6) according to specific range of values for source/destination time to live (10) (11).
38	*ct_flw_http_mthd*	Integer	No. of flows that has methods such as Get and Post in http service.
39	*is_ftp_login*	Binary	If the ftp session is accessed by user and password then 1 else 0.
40	*ct_ftp_cmd*	integer	No of flows that has a command in ftp session.
41	*ct_srv_src*	integer	No. of connections that contain the same service (14) and source address (1) in 100 connections
42	*ct_srv_dst*	integer	No. of connections that contain the same service (14) and destination address (3) in 100 connections.
43	*ct_dst_ltm*	integer	No. of connections of the same destination address (3) in 100 connections
44	*ct_src_ltm*	integer	No. of connections of the same source address (1) in 100 connections
45	*ct_src_dport_ltm*	integer	No of connections of the same source address (1) and the destination port (4) in 100
46	*ct_dst_sport_ltm*	integer	No of connections of the same destination address (3) and the source port (2)
47	*ct_dst_src_ltm*	integer	No of connections of the same source (1) and the destination (3) address in in 100 connections

2.2 Attack classes description

The categories of attack are:

- **Exploits**: code that takes advantage of a software vulnerability or security flaw;
- **Fuzzers**: process of throwing invalid, unexpected or random data as inputs at a computer;
- **Reconnaissance**: attempt to gain information about organization's systems and networks without the explicit permission of the organization;
- **DoS**: attack meant to shut down a machine or network, making it inaccessible to its intended users. This is done, flooding the target with traffic;
- **Generic**: a technique that works against all block-cyphers without consideration against the block-cypher;
- **Shellcode**: a special type of code injected remotely, which hackers use to exploit a variety of software vulnerabilities;
- **Analysis**: a hacker tries to access the same network as you to listen (and capture) all your network traffic;
- **Backdoor**: an attack that exploits any route by which someone can circumvent normal security measures to access a system;
- **Worm**: malware that can propagate or self-replicate from one computer to another without human activation after breaching a system.

Chapter 3

Data pre-processing

The process starts with the reading of the four CSV files containing data on network traffic and inserting them into a *DataFrame*.

The traffic contains both attacks and normal requests and each entry is labeled.

This *DataFrame* will be used for further data cleaning and analysis.

3.1 Removal of not usable features

Then certain features that are not usable in real-world scenarios are dropped, namely *srcip*, *sport*, *dstip*. These features contain IP addresses and port numbers, which can be easily faked using VPNs and are not useful in the real world.

3.2 Feature conversion

After that, some features in the data is converted and parsed.

Ex:

- The feature *ct_ftp_cmd* is parsed to integers, with blank values set to -1.
- The feature *dsport* is converted from hexadecimal values to integers, with missing values set to -1.
- The features *ct_flw_http_mthd* and *is_ftp_login* have missing values set to -1.

Nominal features

Additionally, nominal features are converted: *proto*, *state*, and *service* are enumerated and converted to their corresponding index in the enumeration.

Additionally, the mapping for the conversion of said features is saved for future use: feature conversion of new data needs to be coherent with the training one.

3.3 Data cleaning

Nan values were substituted with -1 and malformed labels were merged (ex: *Backdoor* and *Backdoors*).

After that duplicates were eliminated.

3.4 Data reduction

We analyze the distribution of the data-set based on the presence of attacks and then we analyze the distribution of attack categories in the attack entries.

The analysis shows that the data-set is unbalanced, with a large majority of observations being non-attack instances.

Table 3.1: Dataset distribution

	Occurrencies	**Percentage**
Normal	1958546	95.17%
Attacks	99394	4.82%

This problem will be solved by sampling the "normal requests" once the final cardinality of the "attack requests" is found.

The attack categories, as we can see below, are also very unbalanced.

Table 3.2: Attack categories distribution

	Occurrencies	**Percentage**
Exploits	27599	27.76%
Generic	25285	25.43%
Fuzzers	21642	21.77%
Reconnaissance	13354	13.43%
DoS	5665	5.69%
Analysis	2184	2.19%
Backdoor	1983	1.99%
Shellcode	1511	1.52%
Worms	171	0.17%

We will target this problem by merging the attack classes.

3.4.1 Number of attack classes reduction

The number of attack classes is too high and as we saw, the distribution of them is very unbalanced. To solve this we decided to merge the most similar attack classes.

We decided that similarity between classes is based on how the perform in the data-set and how they relate to each other in the real world. Using this two approaches we will then merge the most similar classes.

To reach this goal we will: - search in the data-set to find out if two or more classes of attacks share some similarities - study the type of attacks to see which classes are more similar in a real world scenario.

Data clustering

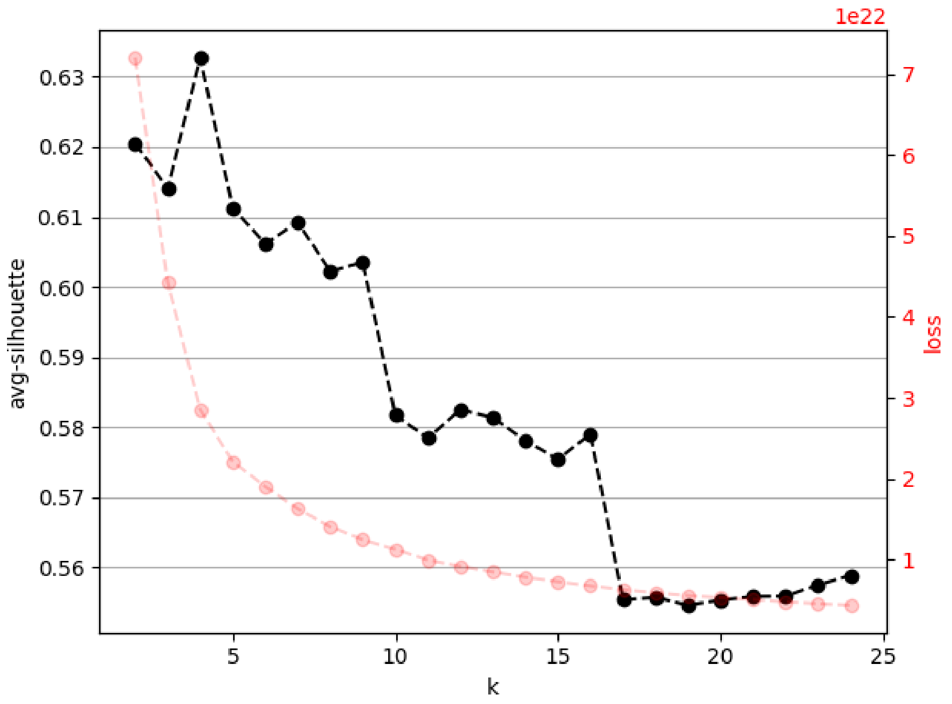
Let's first see the clusterability of the data set:

The clusterability of the data is checked using the hopkins test implemented by the pyclustertend library. Thankfully, the results are close to zero, indicating that the data is very clusterable.

Before using the KMeans algorithm from scikit-learn to perform the clustering, it's necessary to identify the optimal number of cluster.

In order to do that, a cycle was implemented increasing incrementally the number of clusters in every step and using the *silhouette_score* method to evaluate the performance of the clustering.

Figure 3.1: Silhouette-score using k-means sampling by 50-000



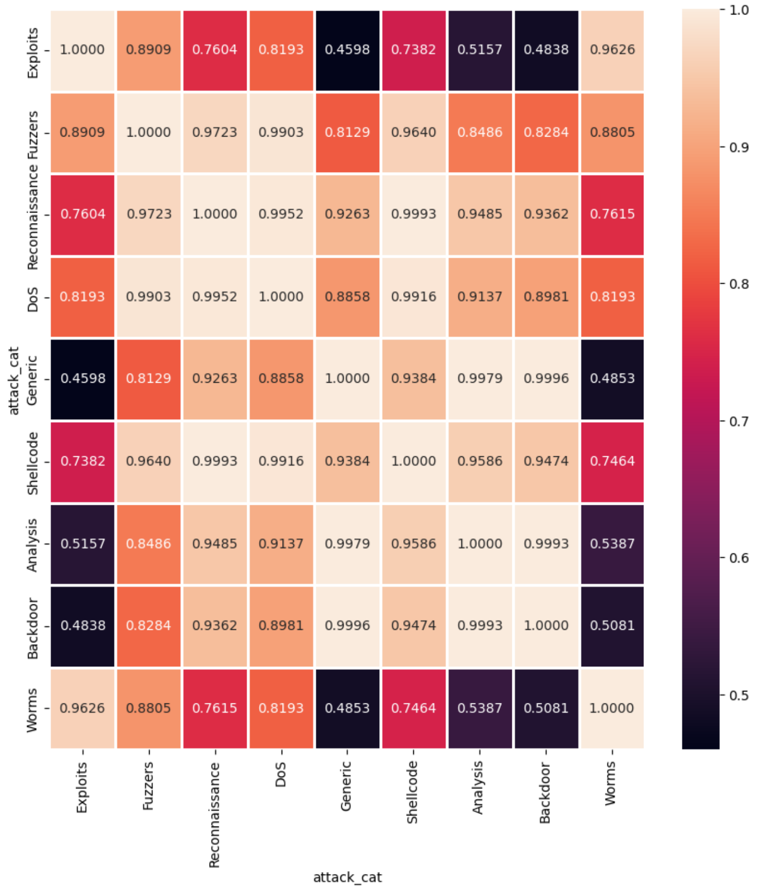
As we can see in the image, the optimal number of clusters is 4. So we re-executed the k-means method without sampling.

For each cluster is then counted the number of elements for each class. And these were the results:

	C0	**C1**	**C2**	**C3**
Generic	89.25%	3.12%	3.92%	3.70%
Fuzzers	41.20%	16.98%	20.74%	21.09%
Reconnaissance	54.14%	13.62%	15.70%	16.54%
Exploits	20.03%	23.80%	27.70%	28.46%
DoS	48.47%	14.95%	18.50%	18.08%
Analysis	81.18%	5.22%	7.01%	6.59s%
Backdoor	85.63%	4.24%	5.24%	4.89%
Shellcode	56.32%	12.71%	15.88%	15.09%
Worms	22.22%	19.88%	38.01%	19.88%

After that the cosine similarity is applied in order to see which attack classes behave similarly (have common cluster presence to one another) so that they could be merged if necessary.

Figure 3.2: Cosine similarity



attack_cat	Exploits	Fuzzers	Reconnaissance	DoS	Generic	Shellcode	Analysis	Backdoor	Worms
Exploits	1.0000	0.8909	0.7604	0.8193	0.4598	0.7382	0.5157	0.4838	0.9626
Fuzzers	0.8909	1.0000	0.9723	0.9903	0.8129	0.9640	0.8486	0.8284	0.8805
Reconnaissance	0.7604	0.9723	1.0000	0.9952	0.9263	0.9993	0.9485	0.9362	0.7615
DoS	0.8193	0.9903	0.9952	1.0000	0.8858	0.9916	0.9137	0.8981	0.8193
Generic	0.4598	0.8129	0.9263	0.8858	1.0000	0.9384	0.9979	0.9996	0.4853
Shellcode	0.7382	0.9640	0.9993	0.9916	0.9384	1.0000	0.9586	0.9474	0.7464
Analysis	0.5157	0.8486	0.9485	0.9137	0.9979	0.9586	1.0000	0.9993	0.5387
Backdoor	0.4838	0.8284	0.9362	0.8981	0.9996	0.9474	0.9993	1.0000	0.5081
Worms	0.9626	0.8805	0.7615	0.8193	0.4853	0.7464	0.5387	0.5081	1.0000

We can clearly see that:

- *Fuzzers* and *DoS*
- *Shellcode* and *Reconnaissance*
- *Analysis* and *Backdoor*

act similarly.

Based on that we would like to merge these classes but *Analysis* and *Backdoor* attacks have nothing in common in the real word, the *Analysis* category in fact has more in common with *Shellcode* and *Reconnaissance* so we will merge it with them. Another category of attack to consider is *Worms* which is, based on our knowledge, similar to *Exploits*.

Attack class merge conclusions

The classes that will be created are:

- *Worms/Exploits*
- *Fuzzers/DoS*
- *Shellcode/Analysis/Recon*

BackDoor will be discarded because despite having a behavior similar to other classes, if merged with them will bring no more information on the attack.

Results of attack classes merge

The result of this operation is the reduction of the attack classes from 9 to 4:

Table 3.3: Merged attack classes distribution

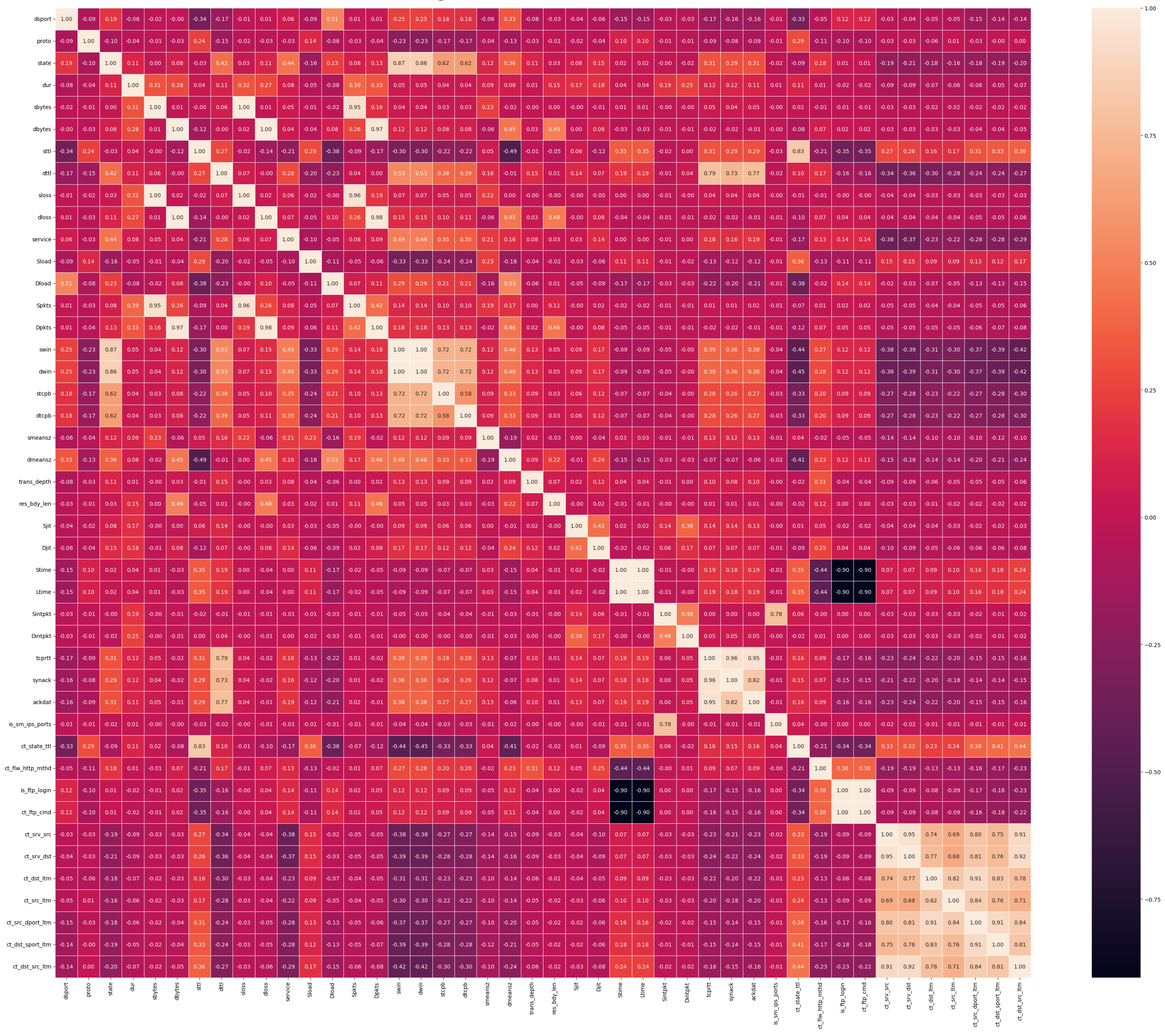
	Occurrencies	**Percentage**
Worms/Exploits	27770	28.50%
Fuzzers/DoS	27307	28.03%
Generic	25285	25.95%
Shellcode/Analysis/Recon	17049	17.50%

3.4.2 Features correlation

Now we analyze the features correlation to see if there are some features which are over-correlated from the remaining ones.

To check this we plotted the correlation matrix:

Figure 3.3: Correlation Matrix

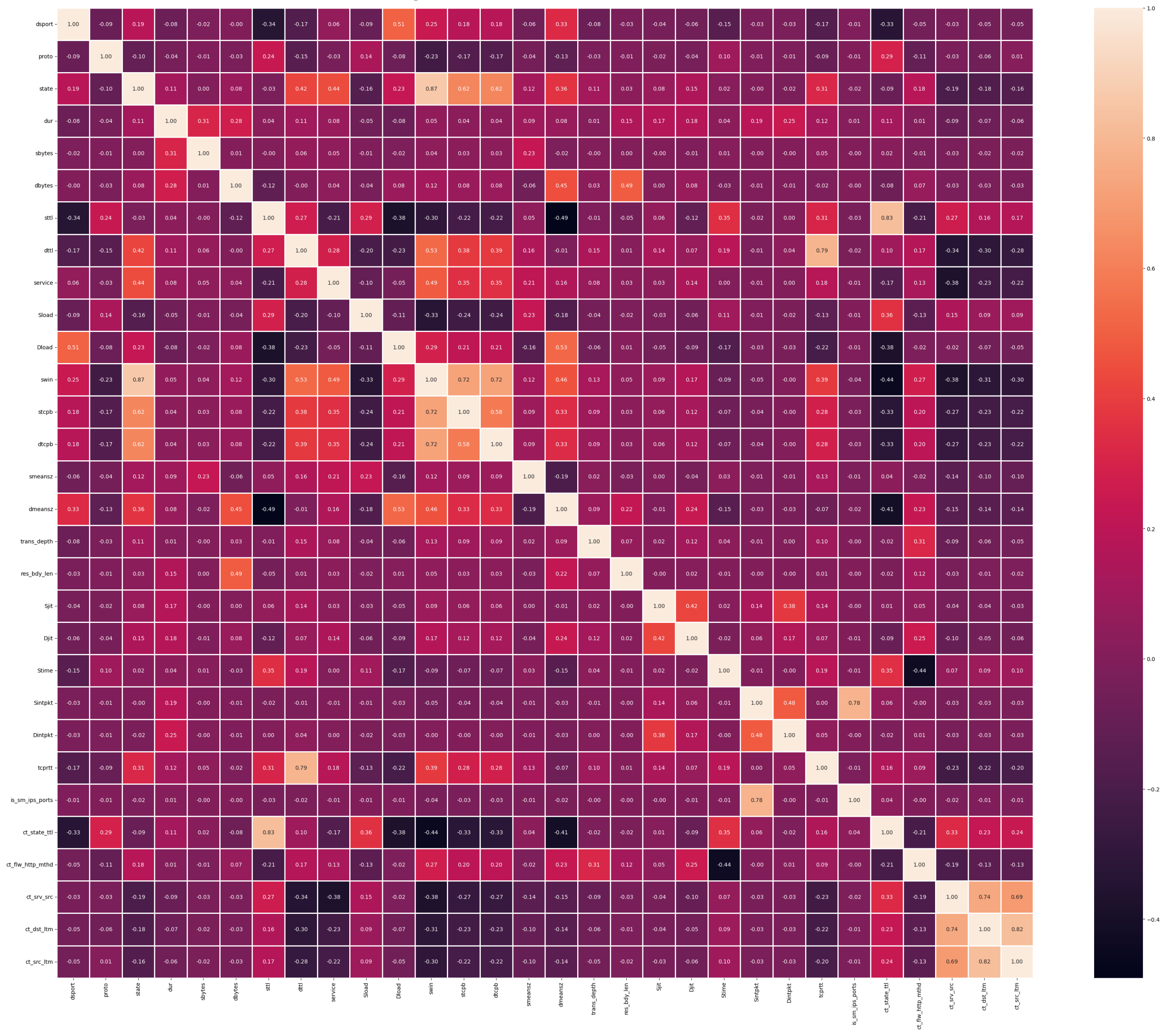


Since the high correlation gives us redundant values, the feature that have correlation grater than 0.90 are dropped to reduce the dimensionality of the data-set.

This was done because over-correlated features bring no information and slow down both the training and the classification process without without improving the accuracy of the models.

This is the new correlation matrix:

Figure 3.4: New correlation Matrix



Chapter 4

Classification

In this section we find the classifier which suits better the problem.

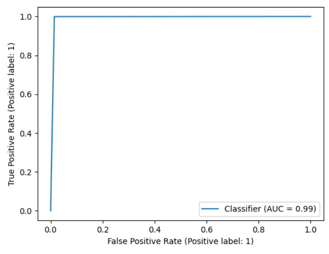
The choice of classifiers is important as it affects the performance of the model. Random Forest, KNeighbors, and MLP are commonly used classifiers that have been proven to have good performance in different problems so we decided to test them.

A dictionary of classifiers including $RandomForestClassifier$, $KNeighborsClassifier$, and $MLPClassifier$ is defined.

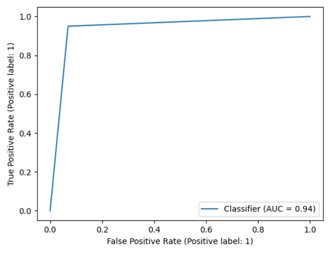
Then, the models of binary and multi-class classifiers are trained.

4.1 Binary classifiers training

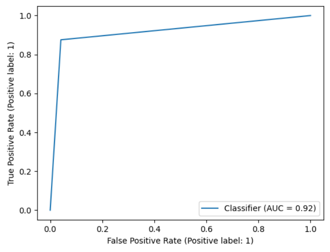
For each classifier we plotted the confusion matrix and the ROC curve. The results that we found can be seen below.



Random forest

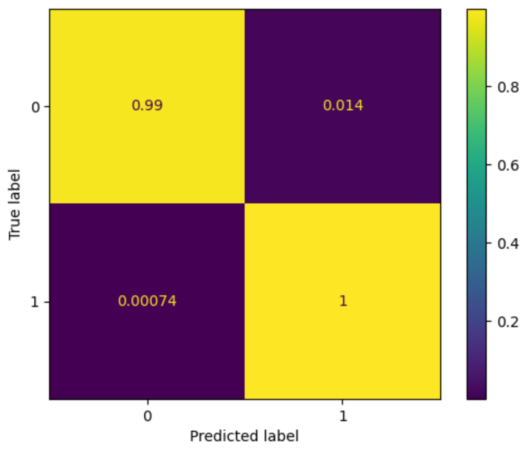


KNeighbors

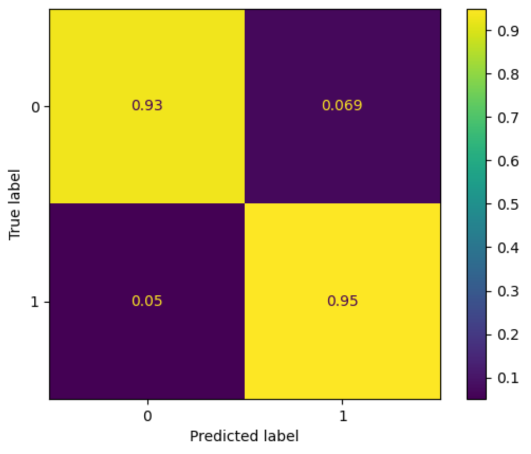


Multi-layer Perceptron

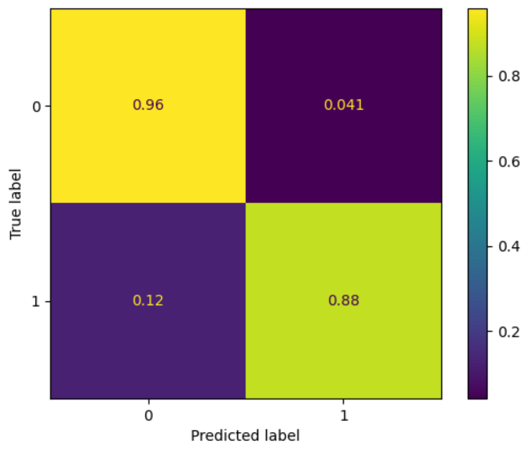
Figure 4.1: ROC curve comparison



Random forest (acc. 99.24%)



KNeighbors (acc. 94.03%)



Multi-layer Perceptron (acc. 91.77%)

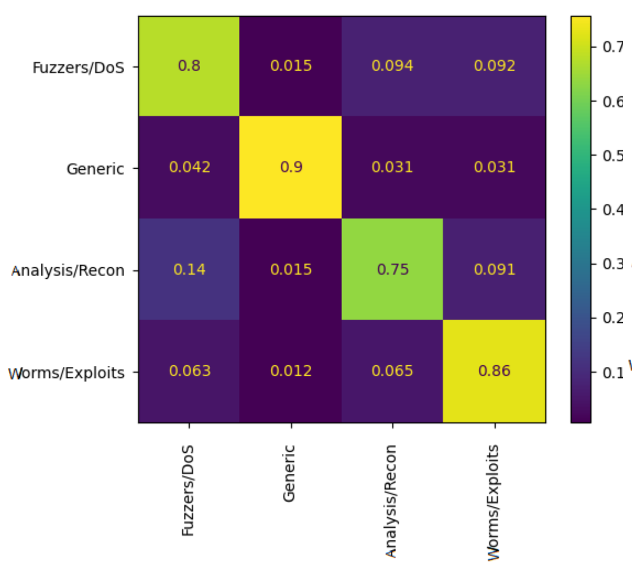
Figure 4.2: Correlation Matrix Comparison

For binary classification the model with the highest *accuracy_score* is the one trained using Random Forest.

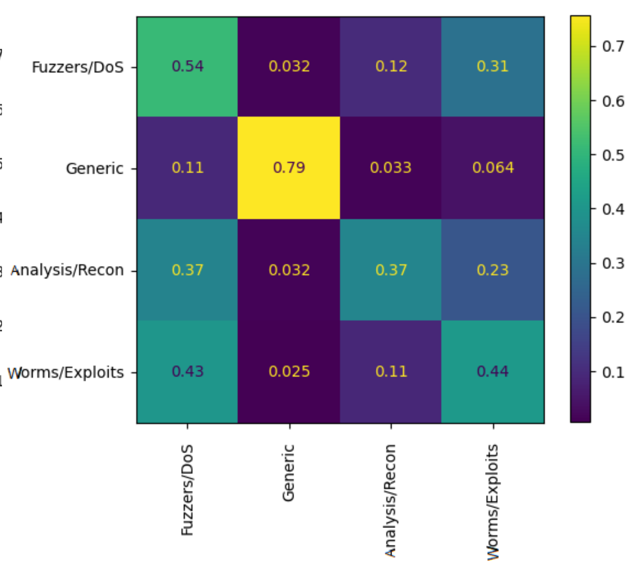
4.2 Multi-class classifiers training

We did the same procedure as the **binary classifier training** and we found these results:

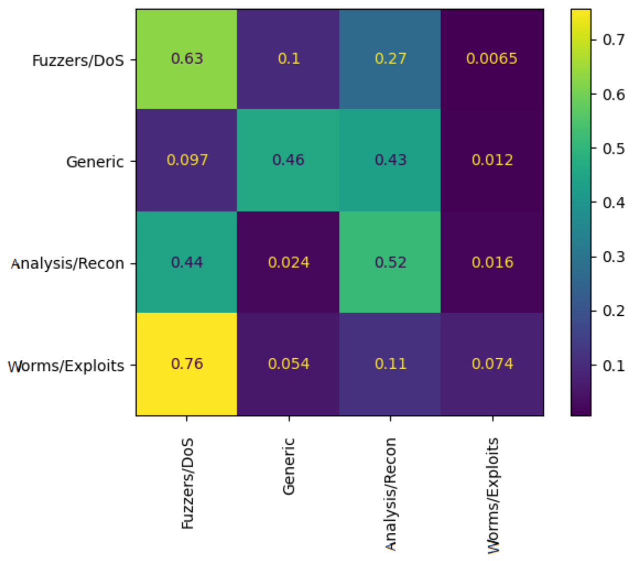
Like before, in the multi-class problem the model with the highest *accuracy_score* is the one trained using Random Forest.



Random forest (acc. 83.69%)



KNeighbors (acc. 54.65%)



Multi-layer Perceptron (acc. 40.58%)

Figure 4.3: Correlation Matrix Comparison

4.2.1 Performance metric decision

The accuracy score is used as a performance metric because it's simple to understand, and it's a good indicator of how well the model is doing in terms of correctly classifying the data.

4.2.2 Model refinment

Once we found out the most performant model we re-trained it 20 times, until we reached the highest *accuracy_score*, which is:

- 99,3394% for the binary classification
- 84,1949% for the multi-class classification

4.2.3 Model storing

Finally, the models were saved using the pickle module to be able to use them later for predictions on new unseen data.

Chapter 5

Real world application

Having our trained models, it's time to test them with real world data.

This data derives from two *pcap*s' requests gotten from recording a network during a "Capture The Flag" event.

The two *pcap*s are:

- **CC22-final-only-bots.pcap** which contains only data which is to consider *normal*
- **CC22-final-also-attacks.pcap** which contains also attacks requests.

5.1 Extraction of non aggregated features

We began by finding the tools to extract the base features we needed from the *pcap* which are:

- *argus* which is used to export from *pcap* to *argus* file format
- *ra* which is a command line tool provided with *argus* that can be utilized to extract features from an *argus* file

A challenge was matching the name scheme of *ra* to the one used in the training data-set, as you can see in the next page.

Table 5.1: ra and training dataset naming schemes

ra	**data-set**	**chosen**	**Used by the models**
saddr	*srcip*	*saddr*	
sport	*sport*	*sport*	
daddr	*dstip*	*daddr*	
–	*dsport*	*dport*	v
proto	*proto*	*proto*	v
state	*state*	*state*	v
dur	*dur*	*dur*	v
sbytes	*sbytes*	*sbytes*	v
dbytes	*dbytes*	*dbytes*	v
sttl	*sttl*	*sttl*	v
dttl	*dttl*	*dttl*	v
sloss	*sloss*	*sloss*	
dloss	*dloss*	*dloss*	
–	*service*	*dport*	v
dload	*Dload*	*dload*	v
sload	*Sload*	*sload*	v
spkts	*Spkts*	*spkts*	
dpkts	*Dpkts*	*dpkts*	
swin	*swin*	*swin*	v
dwin	*dwin*	*dwin*	
stcpb	*stcpb*	*stcpb*	v
dtcpb	*dtcpb*	*dtcpb*	v
smeansz	*smeansz*	*smeansz*	v
dmeansz	*dmeansz*	*dmeansz*	v
trans	$trans_depth$	*trans*	v
psize	*res_bdy_len*	*psize*	v
sjit	*Sjit*	*sjit*	v
djit	*Djit*	*djit*	v
stime	*Stime*	*stime*	v
ltime	*Ltime*	*ltime*	
sintpkt	*Sintpkt*	*sintpkt*	v
dintpkt	*Dintpkt*	*dintpkt*	v
tcprtt	*tcprtt*	*tcprtt*	v
synack	*synack*	*synack*	
ackdat	*ackdat*	*ackdat*	

5.2 Data processing

Which consisted of:

- calculation of the aggregated features
- preprocessing

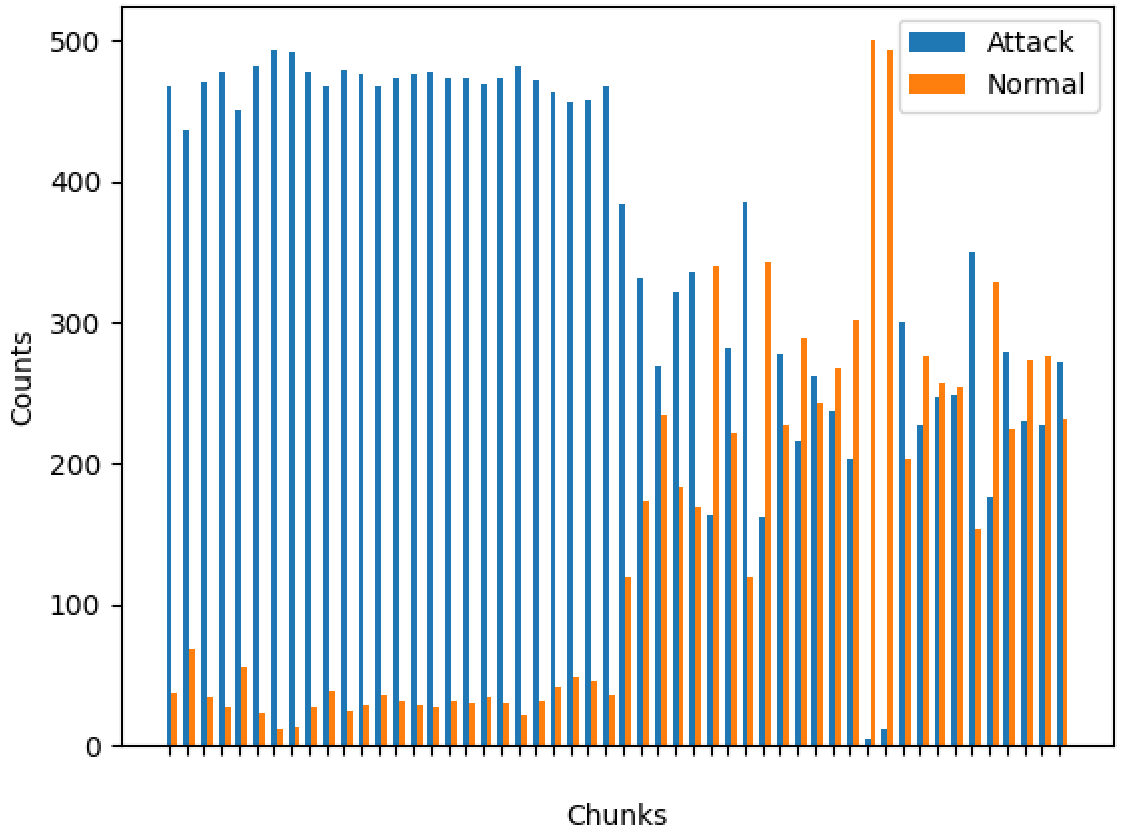
5.2.1 Data pre-processing

We've done the same pre-processing done on the training data-set using the previously calculated maps to map the nominal features into numbers.

5.3 Classification

We've tried to use the trained model directly on the **CC22-final-only-bots.pcap** file which from now on will be called "baseline", and these were our results:

Figure 5.1: Binary classification of the baseline without scaling



As we can see from the image, our binary model detected multiple attacks, which is in contrast to our knowledge on the data-set. This means that we have a lot of false-positives (we measured a percentage of 71.15%)

5.3.1 Scaling

To solve this problem we realized that the numerical features had to be scaled proportionally to the training data-set

Our model in fact is trained with absolute numerical features which cannot be the same for each traffic we analyze.

Our solution is to use a known "baseline" of traffic which contains only "normal" packages of the service we want to analyze.

And then calculate the $scaling_factor$ using the mean value of each numerical feature, both from the training data-set and the "baseline", remembering to remove the attacks from the training data-set while doing so.

$$scaling\_factor[feature] = \frac{mean(baseline[feature])}{mean(dataset[feature])}$$

These are the $scaling\_factor$ obtained:

Table 5.2: Scaling factors

feature	**scaling_factor**
dur	0.10890239321483493
sbytes	0.05937458910275042
dbytes	0.019195294402651504
Sload	0.003990302362924578
Dload	0.009235022017737944
swin	718.3728728344787
res_bdy_len	0.12638913034817256
Sjit	-0.00018074807431660O5
Djit	-0.0003066024871620531
Stime	0.3051080341915523
Sintpkt	-0.0015412293513965111
Dintpkt	-0.003723888346134258
tcprtt	3.2800416921042737
smeansz	0.20426072955147842
dmeansz	0.10877900200409872
trans_depth	2.755887300252313
ct_state_ttl	0.00029806259314456036
ct_flw_http_mthd	-0.0
ct_srv_src	3.1023816067553094
ct_dst_ltm	7.142034691425571

Once the scaling_factors were calculated we used them to multiply each features.

5.3.2 Scaled classification

We then used our model to predict both the "baseline" and the *pcap* which is known to have attacks in it, scaling the features with their *scaling_factor* these were our results:

Figure 5.2: Binary classification of the baseline with scaling

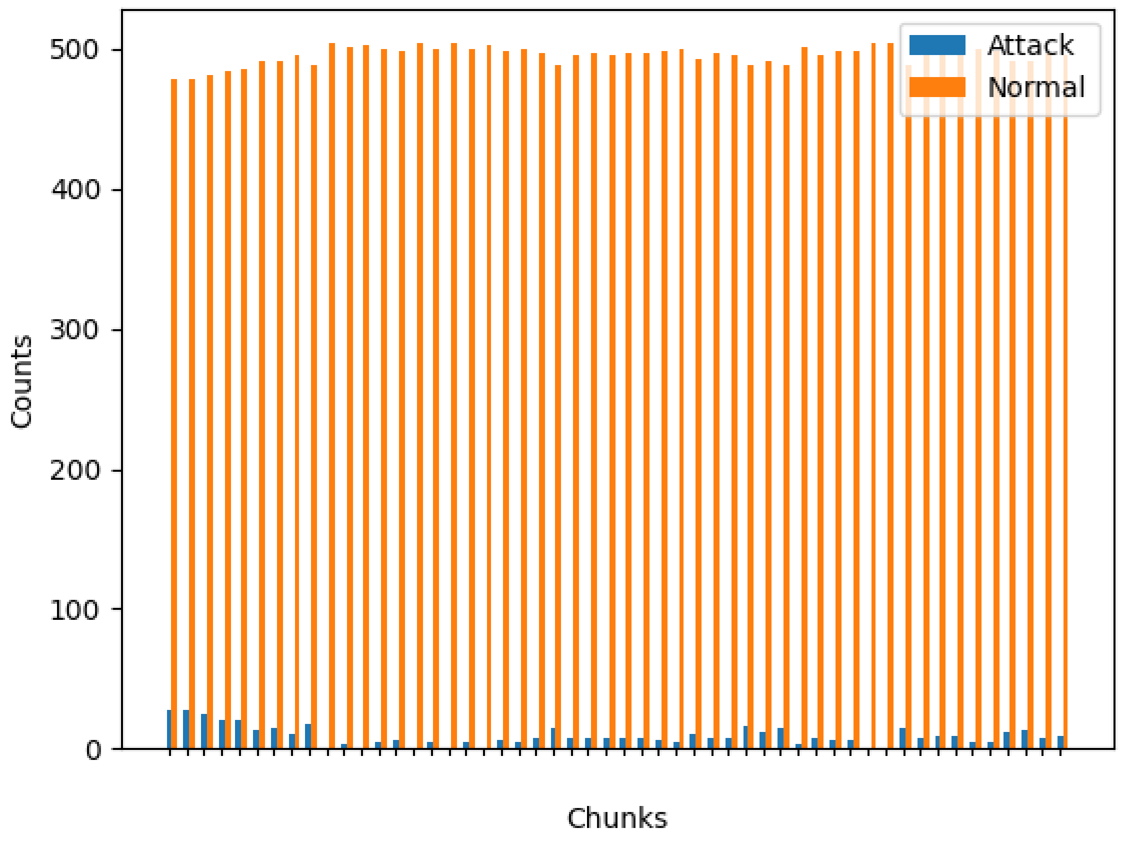
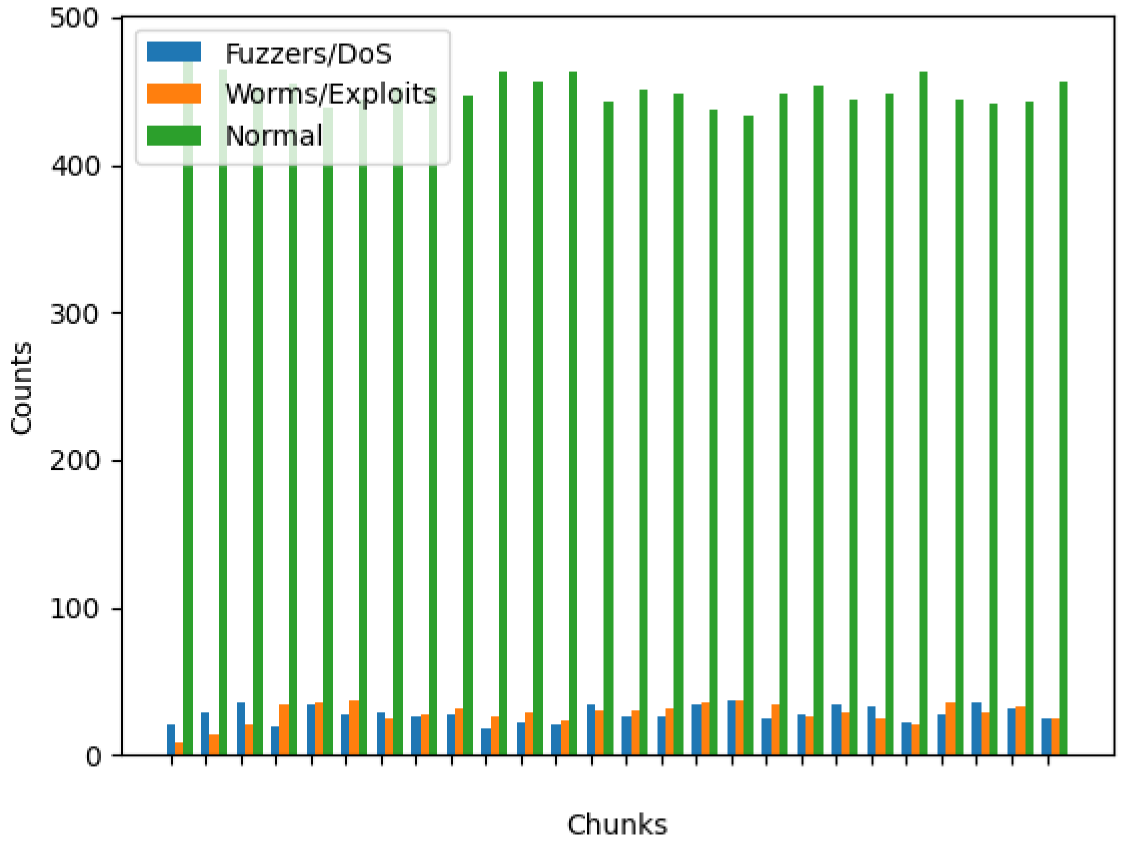


Figure 5.3: Multi-class classification



We can see that in the scaled "baseline" we found only a few false attacks, the false-positive percentage is only 1.75%.

We cannot say more about the accuracy of the classifiers in the other *pcap*, because we have no information on it.

Chapter 6

Conclusion

We succeeded in training a model that works in a scenario different from the training one.

Despite that we have no knowledge on the *pcap* containing also attacks we can affirm that:

- the binary model found a low percentage of attacks in the "baseline" *pcap* (that surely corresponds on a low percentage of false positives)
- the binary model found some attacks in the *pcap* we knew had some, in a magnitude much higher respect to the "baseline" one
- the classes of attacks found by our multi-class model are coherent with the one we expected to find.

The models that were trained despite having some accuracy problems, could be used in a real world scenario with other tools to automatically block malicious users, or at least they could be used to gain insights useful to analyze a huge traffic of data.

Bibliography

[1] Nour Moustafa and Jill Slay. “UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 Network Data Set)”. In: *2015 Military Communications and Information Systems Conference (MilCIS)* (2015). DOI: 10.1109/milcis.2015.7348942.