School of Computing and Information Systems
The University of Melbourne

COMP90073 Security Analytics, Semester 2 2022

Assignment 2

Detecting cyberattacks in network traffic data

Student name	**Minh Tu Nguyen**
Student ID	**1321154**

STUDENT DECLARATION

I declare that:

- This assignment is my own original work, except where I have appropriately cited the original source.
- This assignment has not previously been submitted for assessment in this or any other subject.

For the purposes of assessment, I give the assessor of this assignment the permission to:

- Reproduce this assignment and provide a copy to another member of staff; and
- Take steps to authenticate the assignment, including communicating a copy of this assignment to a checking service (which may retain a copy of the assignment on its database for future plagiarism checking).

Contents

Task I

1. Introduction

Analysing network traffic is no easy task, as with the increase of internet services, the amount of traffic per day is extremely large. It is nearly impossible for human to supervise all incoming and outgoing traffic. It is of much importance to have an automatic or autonomous solution that helps isolate cyber attack traffic. With the help of machine learning and deep learning, many sophisticated and efficient methods have been proposed. There are two main approaches in using machine learning or deep learning for detecting cyber attack traffic, which are supervised and unsupervised algorithms, each has its own advantages and disadvantages. In the domain of cyber security, the amount of normal data is usually significantly bigger, which will cause the problem of unbalanced data. Moreover, supervised learning algorithms might be efficient when detecting a “known” attack, however, for “unknown” attack, or anomalies, they might suffer. Exhaustive and challenging labelling for data is another reason why some researchers and security analysts prefer unsupervised algorithms.

Task I of this report will discuss using relatively simple unsupervised machine learning algorithms to detect network anomalies. Task I of the report is structured as follows: Section 2 gives an overview of the test dataset, Section 3 performs some feature engineering and feature selection techniques, Section 4 conducts experiments of two simple models: IsolationForest and LOF on 7 sets of selected features, discuss the scores and post-processing technique to improve the detection rate.

It is noteworthy that the Label field of the test dataset is not touched during any phase except the final report of the result, not during the analysis, training or evaluating.

2. Test set overview with Splunk

Ingest the test dataset into Splunk by placing the csv file to the lookups folder of Splunk_ML_Toolkit app folder and use `inputlookup` command in Splunk ML app to search, or adding folder to Data Inputs and use `source` command in Search & Report app to search.

Test set records `764723` flows, from `2021-08-12 20:56:02` to `2021-08-12 23:16:19` and from `2021-08-13 00:48:20` to `2021-08-13 01:18:26`.

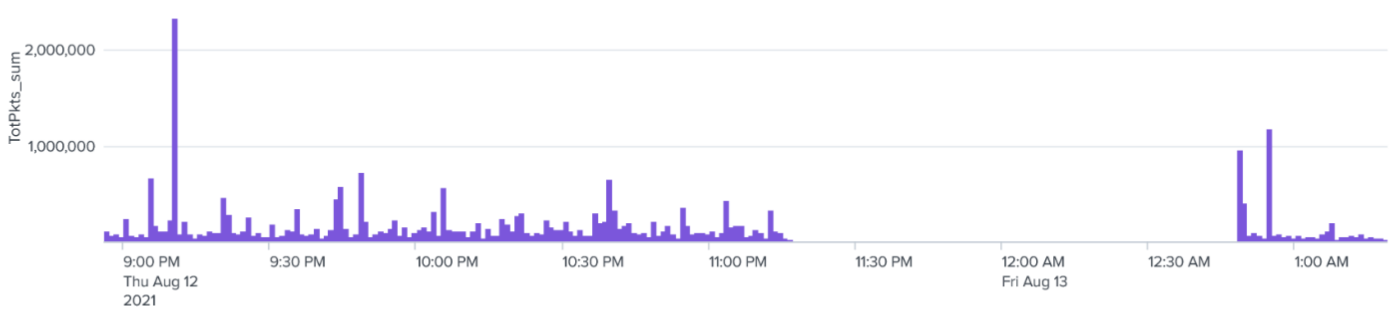


Fig I-1. Total Packets per time span

(1) Top Protocol

```
| inputlookup t.csv | stats count by Proto | sort -count
```

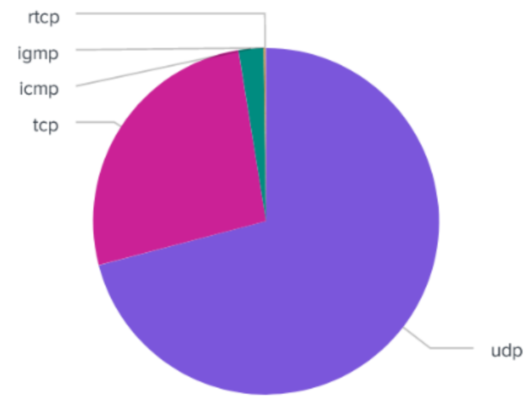


Fig I-2. Top Protocol

(2) Top State

```
| inputlookup t.csv | stats count by State | sort -count
```

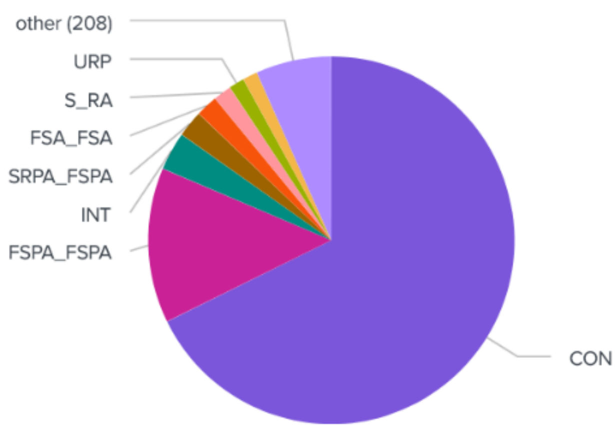


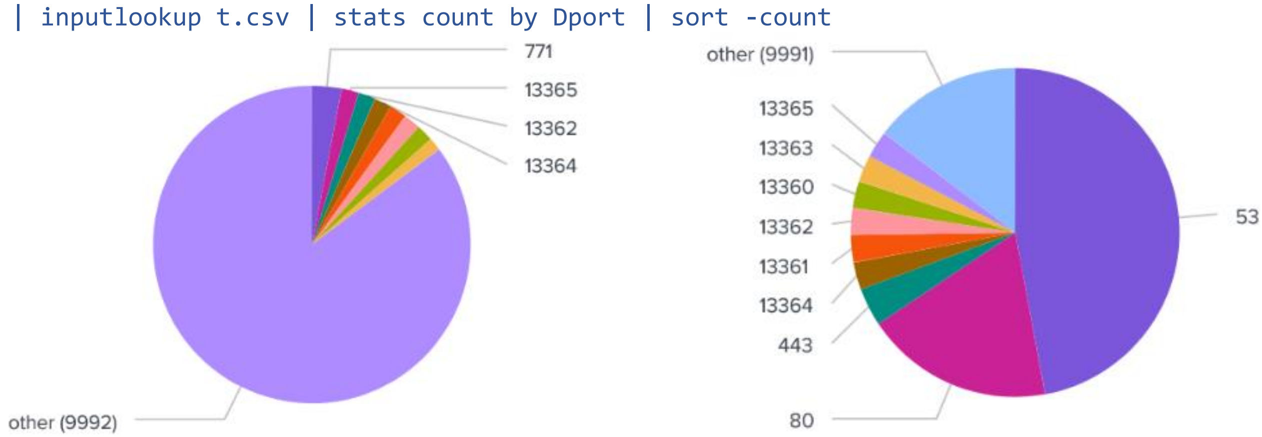
Fig I-3. Top State

Table I-1. Top 10 State

#	State	count
1.	CON	517871
2.	FSPA_FSPA	104486
3.	INT	25966
4.	SRPA_FSPA	17877
5.	FSA_FSA	14739
6.	S_RA	12307
7.	URP	10414
8.	S_	9972
9.	FSRPA_FSPA	6731
10.	FSPA_FSRPA	4413

(3) Top Sport, Dport

```
| inputlookup t.csv | stats count by Sport | sort -count
| inputlookup t.csv | stats count by Dport | sort -count
```



Top Sport *Top Dport*

Fig I-4. Top Sport & Top Dport

(4) Top Conversation

```
| inputlookup t.csv
| eval Conversation=SrcAddr."|".DstAddr
| stats count by Conversation | sort -count
```

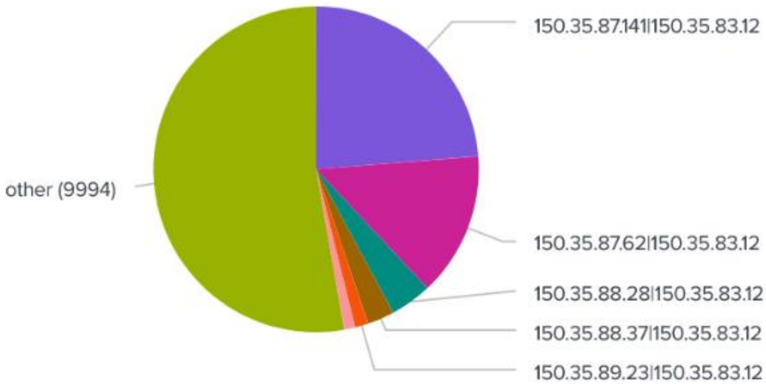


Fig I-5. Top Conversation

(5) TotBytes by Proto per time span (timespan = 1min)

```
| inputlookup t.csv
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart sum(TotBytes) as TotBytes_sum by _time, Proto
```

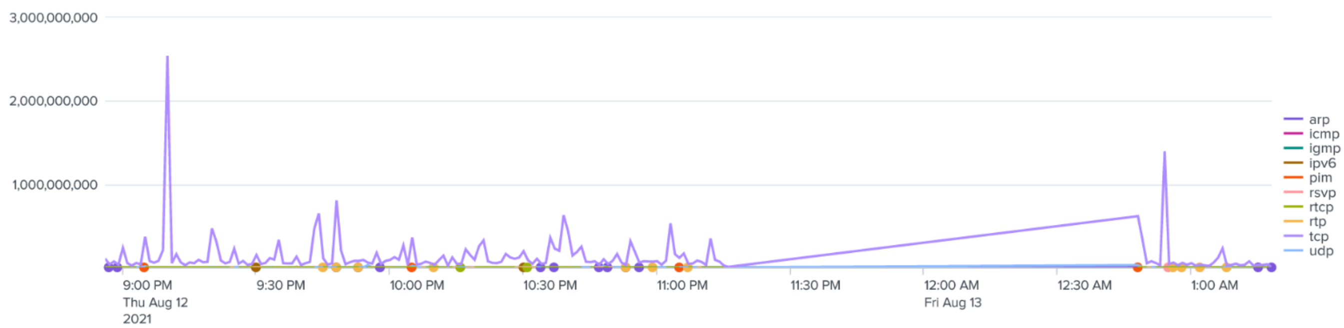


Fig I-6. Total bytes transferred by Proto per time span (timespan = 1min)

(6) TotBytes by Conversation per time span (timespan = 1min)

```
| inputlookup t.csv
| eval Conversation=SrcAddr."|".DstAddr
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart sum(TotBytes) as TotBytes_sum by _time, Conversation
```

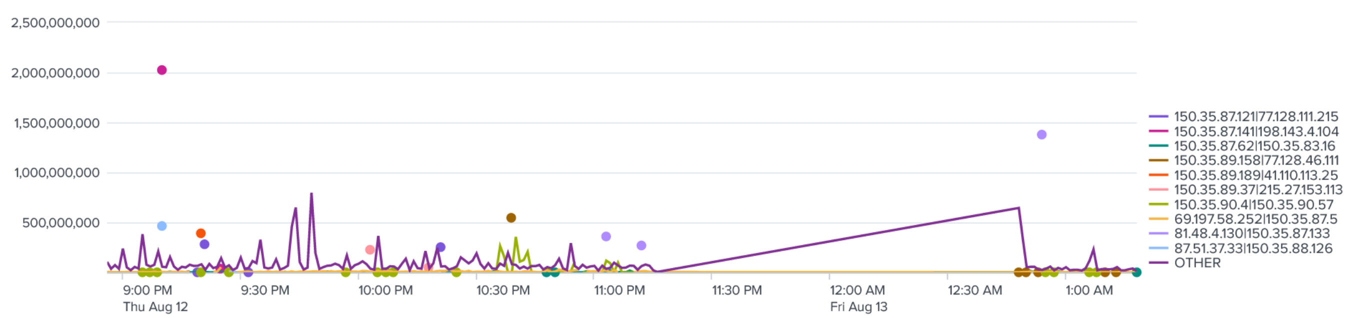


Fig I-7. Total bytes transferred by Conversation per time span (timespan = 1min)

(7) Total SrcBytes per time span (timespan = 1min)

```
| inputlookup t.csv
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart sum(SrcBytes) as SrcBytes_sum by _time
```

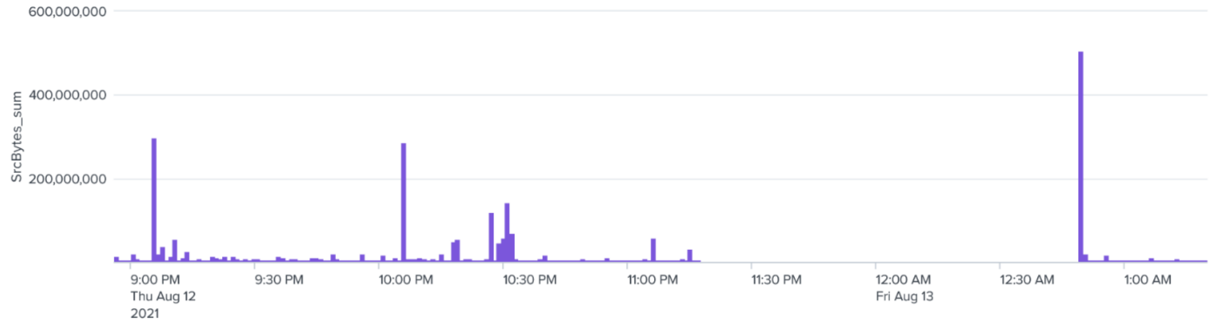


Fig I-8. SrcBytes per time span (timespan = 1min)

(8) Total TotBytes per time span

```
| inputlookup t.csv
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart sum(TotBytes) as TotBytes_sum by _time, Conversation
```

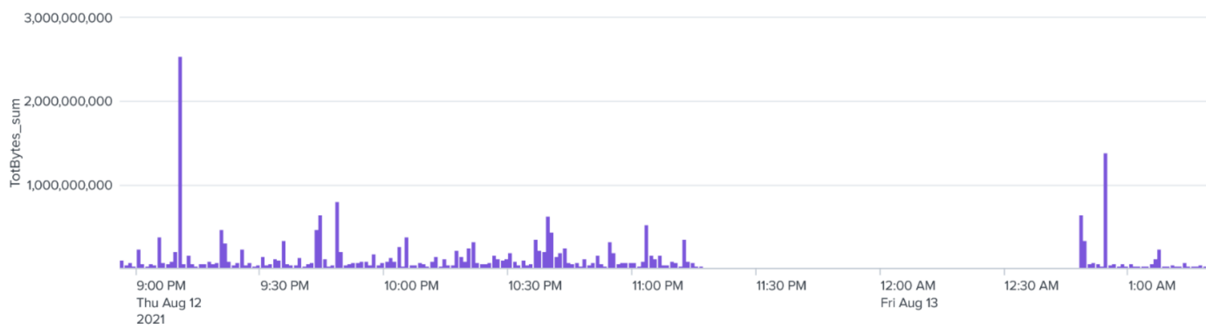


Fig I-9. TotBytes per time span (timespan = 1min)

(9) Total TotPkts per time span

```
| inputlookup t.csv
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart sum(TotPkts) as TotPkts_sum by _time, Conversation
```

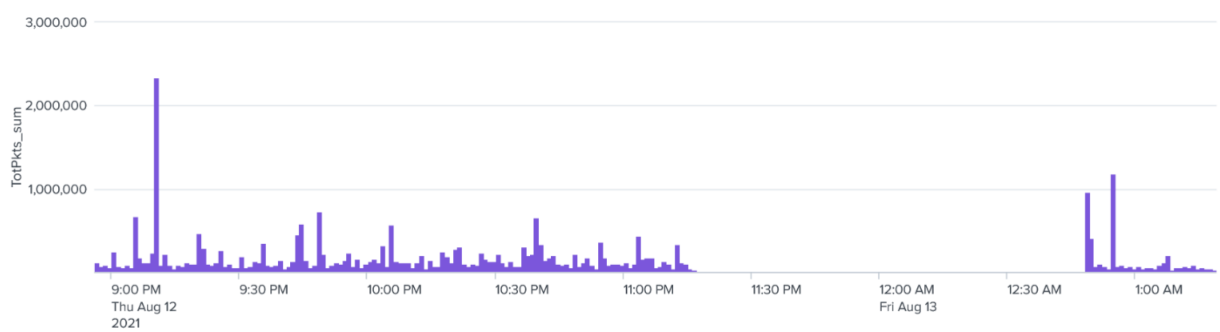


Fig I-10. TotPkts per time span (timespan = 1min)

(10) Mean BytesPerPkt by Conversation per time span

```
| inputlookup t.csv | eval Conversation=SrcAddr."|".DstAddr
| eval BytesPerPkt=TotBytes/TotPkts
```

```
| eval _time=strptime(StartTime,"%Y-%m-%d %H:%M:%S.%6Q") | bin _time span=1m
| chart avg(BytesPerPkt) as BytesPerPkt_avg by _time
```

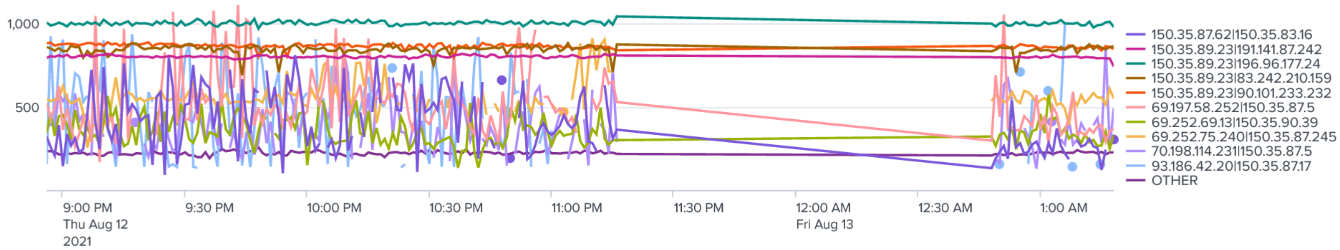


Fig I-11. Mean BytesPerPkt by Conversation per time span (timespan = 1min)

3. Features engineering

This section aims to generate set of features used for anomaly detection methods. All features after this step are listed in Table I-2.

Table I-2. Original features

	Feature	Type		Feature	Type
1	Dur	float64	2	Proto	object
3	SrcAddr	object	4	Sport	float64
5	Dir	object	6	DstAddr	object
7	Dport	float64	8	State	object
9	sTos	float64	10	dTos	float64
11	TotPkts	float64	12	TotBytes	float64
13	SrcBytes	float64			

All generated features are listed in Table I-3. (53 features)

Table I-3. All features after generation

	Feature					
Original	Dur	TotPkts	TotBytes	SrcBytes	sTos	dTos
	Sport	Dport				
Calculated	PktsPerSec	BytesPerSec	BytesPerPkt	SrcBytesPerSec	DstBytes	DstBytesPerSec
State one-hot encoded	State_CON	State_alltcp	State_INT	State_S_	State_A_	State_ECO
	State_RED	State_REQ	State_ECR	State_TXD	State_URFIL	State_R_
	State_URP	State_URHPRO	State_URN	State_RSP	State_URH	State_other
Generated and one-hot based on State	Flag_nan	Flag_S	Flag_A	Flag_R	Flag_P	Flag_F
Proto one-hot encoded	Proto_udp	Proto_tcp	Proto_icmp	Proto_rtp	Proto_rtcp	Proto_igmp
	Proto_arp	Proto_other				
Generated and one-hot based on Dport	Service_80	Service_443	Service_21	Service_22	Service_25	Service_6667
	Service_other					

3.1. Features generation / encoding

The code for this section is in Python, file `vv1-1.ft_gen.__train__.ipynb` and `vv1-1.ft_gen.__testval__.ipynb`. Some features can be generated using splunk command.

Machine learning algorithms require all features being represented as numerical. Hence, we need to find a way to encode string fields, as these fields contain much information. In fact, in Section 4, I conduct an experiment on numeric fields only to prove that using the original numeric features to detect anomalous behaviors does not give as good result using the same algorithm.

3.1.1. Numeric fields

(1) PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt

As discussed in assignment 1, a bot can have repeated behaviour, which means a lot of flows from bots might have similar number of packets per second or number of bytes per second. The following 4 features can demonstrate the relation between `TotPkts`, `TotBytes`, `SrcBytes` and `Dur` (Duration); as well as between `TotBytes` and `TotPkts`.

- `PktsPerSec` (number of packets per second)
- `BytesPerSec` (number of total bytes (both direction) per second)
- `SrcBytesPerSec` (number of total bytes (from source to destination) per second)
- `BytesPerPkt` (number of bytes (both direction) per packet)

Use the following Splunk command or Python code to generate the aforementioned features:

```
| inputlookup t.csv | eval Conversation=SrcAddr."|".DstAddr
| eval PktsPerSec=TotPkts/Dur
| eval BytesPerSec=TotBytes/Dur
| eval SrcBytesPerSec=SrcBytes/Dur
| eval BytesPerPkt=TotBytes/TotPkts
```

(2) sTos and dTos

`sTos` and `dTos` indicate the type of service on the host and destination device, respectively. Values of `sTos` and `dTos` seen in the dataset are: NaN, 0, 1, 2, 3. `sTos` can also have value of 192. ToS (Type of Service) is made up of 8 bits, whereas the first 3 bytes are IP precedence and used to define a precedence, and the last 6 bits declare type of service. IP precedence is used to specify class of service (CoS) for each packet. This CoS value will be mapped with the network policy to determine bandwidth allocation and congestion management strategy for the packet. ToS value of 192 is therefore equivalent to 011000000, which means the CoS is 011 (flash), with normal delay, normal throughput, and normal reliability. Normally, the higher value of ToS indicates a more important packet. The ToS assignment is usually defined as close to the edge of the network or the administrative domain as possible, and this value should always be overridden by the network policy within the network, rather than set in the network client. For this reason, we can safely assume high value of ToS (at least the first 3 bits is different from routine class: 000) indicates high priority packets assigned by the network policy. It means these values should not be of our concern considering detecting bots or anomalies. Therefore, I drop all the records with `sTos=192`.

3.1.2. String fields

Due to the way I deal with string fields before one-hot encoding, I choose a more familiar language to me, which is Python, to preprocess data.

(1) State

The state of a flow represents the status of protocol and flags triggered. For example, for an ICMP connection flow, the state can signify the returning status of the ICMP response; for a TCP connection, the state indicates the direction of the flow and the flags triggered in each direction. For example,

- CON = Connected (UDP);
- INT = Initial (UDP);
- URP = Urgent Pointer (UDP);
- F = Flag F (FIN) triggered (TCP);
- S = Flag S (SYN) triggered (TCP);
- P = Flag P (Push) triggered (TCP);
- A = Flag A (ACK) triggered (TCP);
- R = Flag R (Reset) triggered (TCP);
- FSPA = All flags (FIN, SYN, PUSH, ACK) triggered (TCP);
- REQ, UNK, URFHIL,... = state of ICMP flows
- ...

Symbol _ in state indicates the direction of the flow. For example, `S_` means the flow has direction `forward (->)` with flag `S` triggered from `SrcAddr` to `DstAddr` (with no packet going back from `DstAddr` to `SrcAddr`, as there is no flag triggered in returning direction). Therefore, this field in fact can convey information of both `Proto` and `Dir` fields. Intuitively, we want to encode this field in a way that there is no need to encode `Proto` and `Dir` fields anymore.

Since with tcp flows, there are many values for `State`, I break this down into 5 `Flag_X` fields, with the value of `Flag_X = 1` if flag X is triggered in one direction, and `Flag_X = 1` if flag X is triggered in both directions. X are `[S, A, P, R, F]` – all tcp flags.

However, the range of remaining values of `State` is still very large, due to the fact that one protocol can have multiple response status. Therefore, I choose only top common values of State in train set (values that have over 100 records), all the other values are replaced with `other`, then one-hot encode this field. It is noted that before choosing top common values, all values of tcp protocol are replaced with `alltcp`, except for those values of the format: `X_` (for example, `S_` or `A_`). Using knowledge from network attack scenarios, `S_` flows or `A_` flows can indicate that there are only `SYN` or `ACK` packets within the flow, which can be a signal of flooding or scanning attack, hence, we do not want to omit this significant information.

It is also noteworthy that we use data from train set to encode fields but do not touch the Label at all. This step of retrieving top common values of State in train set ensures we keep most common values and assume they are normal behaviours.

One problem with encoding all other uncommon values to `other` is that there may be many infrequent records now become frequent (with the value of `other`), and more importantly, these infrequent `State` values can be of different protocols' responses. Therefore, we still need to encode the field `Proto`, as intuitively, if the record has the same value of `Proto`, yet the value of `State` is `other`, it might be considered abnormal.

(2) Proto

There are many protocols, we should concern only with the most common protocols. Here I choose to keep the top common values of `Proto` in train set (values that have over 100 records), all the other values are replaced with `other`, then one-hot encode this field.

(3) Dport to Service

As discovered in Assignment 1, `Dport` and `Proto` can be a pattern of an attack. However, `Dport` range can be large (0-65535), and the value of `Dport` does not have "magnitude" meaning, in fact, all 65535 values shall have the same meaning, which should be treated as categorical fields. However, it is infeasible to one-hot encode all 65535 values. Instead, I choose only common services to encode (for example, `Dport 80 = HTTP Service`, `Dport 25 = SMTP Service`, etc.). I keep 6 values of `Dport` for mapping to `Service`: `[80,443,21,22,25,6667]`, all other values are mapped to value `other` for `Service` field.

It should be noted here that the field `Service` is used in addition with the field `Dport` and `Sport`. As using only `Service` field will omit much information if the service is hosted on custom ports (for example, port 8080 can also be used to serve web service etc.). Moreover, this tactic cannot be applied for `Sport`, as when initialising outgoing request, the machine can use random ports. And although `Dport` or `Sport` values do not have "magnitude" meaning (regarding the greater value has more weight), but the magnitude of these values can indicate something. Port numbers can be in the range of 0 to 65353. Port numbers from 0 to 1023 are reserved for well-known services or applications. Port numbers from 1024 to 49151 are called registered ports, they are not assigned but can be registered to prevent duplication. Port numbers from 49152 to 65535 are dynamic ports or ephemeral ports, as these ports are normally not assigned for public services or registered, the operating system uses these ports as temporary ports to return traffic. This means a greater value of `Dport` shall indicate a more abnormal service (if the direction of traffic is from client IPs to server IPs), or a repeatable value of `Sport` with similar value of `PktsPerSec`, `BytesPerSec`, etc. can indicate a C2 connection via a reverse shell (a reverse shell is running an opening an "abnormal" port on an intra machine).

3.2. Features analysis

This section analyses all the features including newly generated ones on validation set.

The code for this section is in Python, filename `vv2-1.ft_sel_1.ipynb`.

3.2.1. Categorical features

(1) Proto

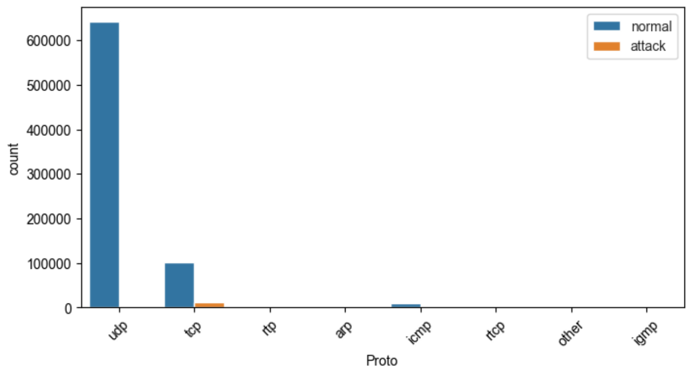


Fig I-12. Number of Proto values in normal and attack flows

- The majority of the traffic protocol are udp and tcp.
- Attacks' protocols are tcp and udp with the counts of: 11178 attack flows with tcp protocol, and 330 with udp protocol. The ratio of attack to normal flows is greater with tcp than with udp.

(2) Dir

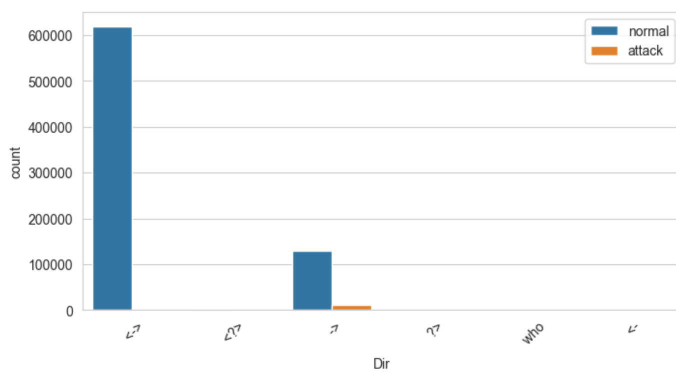


Fig I-13. Number of Dir values in normal and attack flows

- Majority of the flows are bidirectional or forward.
- Attacks can be either bidirectional or forward, with the counts: 11178 forward and 330 bidirectional. 11178 forward flows are tcp flows, and 330 bidirectional flows use udp protocol.

(3) Service

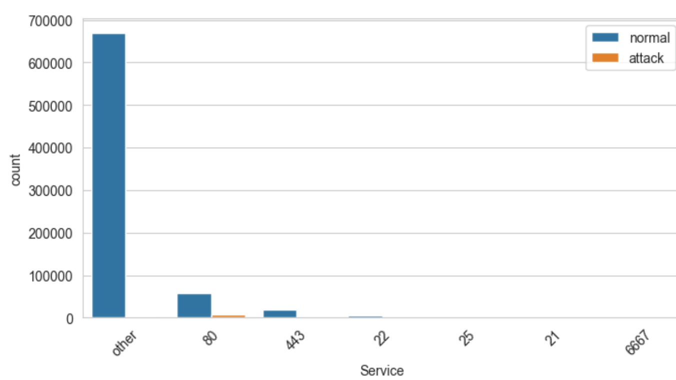


Fig I-14. Number of Service values in normal and attack flows

- Number of attack flows with respect to `Service` is: 8162 for Service 80 (HTTP), 861 for Service 25 (SMTP), 431 for Service 443 (HTTPS), 33 for Service 6667 (IRC) and 2021 for other services.

(4) State

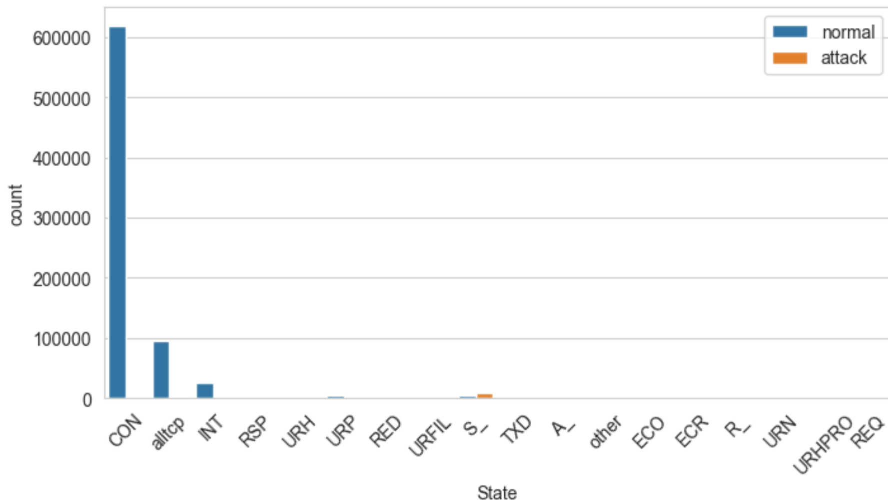


Fig I-15. Number of State values in normal and attack flows

- Majority of attacks having `S_` state. 8227 attack flows have `S_` state, 2951 have `alltcp` state, and 330 having `CON` state. These 330 `CON` flows are 330 `udp` flows that query DNS and establish UDP connection.

(5) sTos and dTos

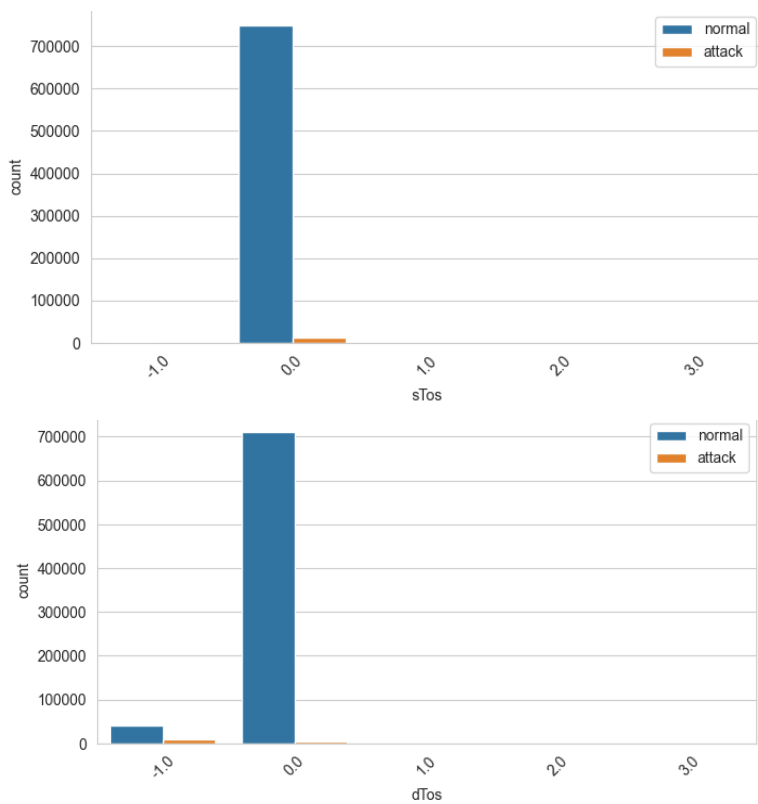


Fig I-16. Number of sTos and dTos values in normal and attack flows

- Attacks have `sTos` of 0 only.
- Attacks can have `dTos` of 0 or NaN (which is replaced with -1). Number of attack flows having `dTos` of 0 and -1 are 3355 and 8153 respectively.

3.2.2. Numeric features

(6) Sport, Dport

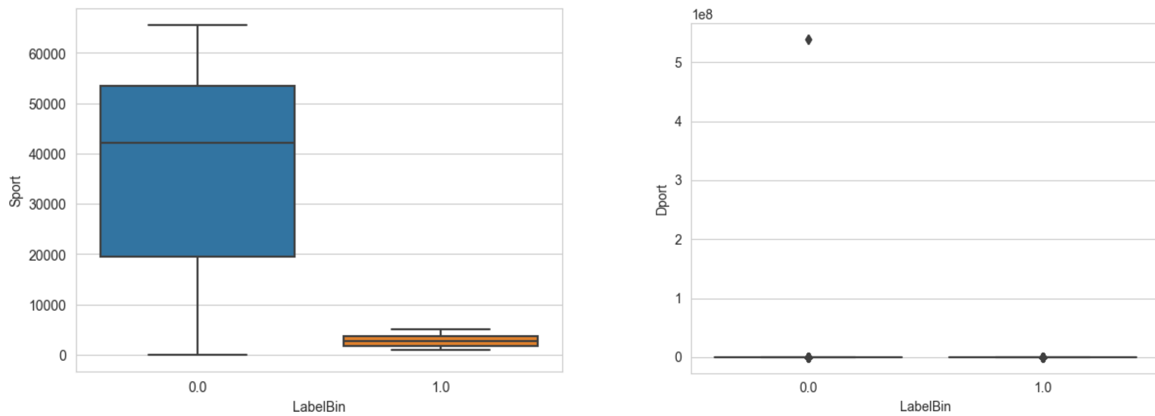
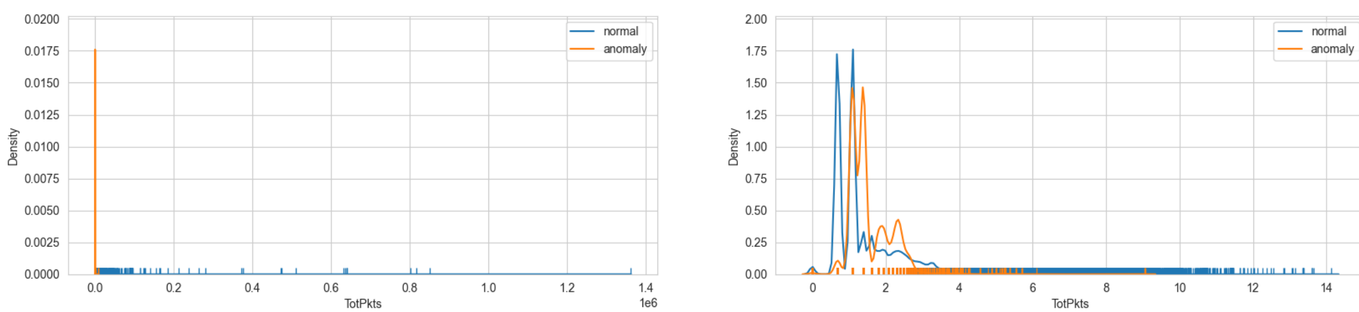


Fig I-17. Bin plot of Sport and Dport

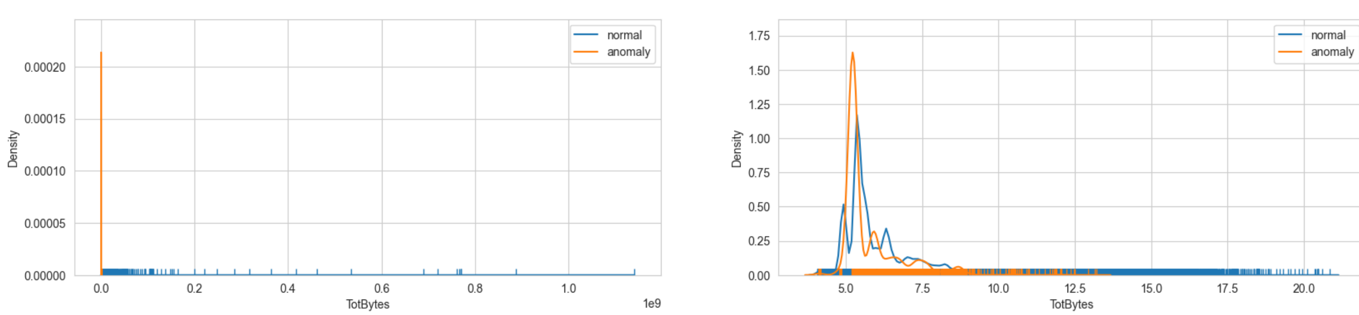
- Attack flows have smaller range of `Sport` and `Dport`.
- Most common value of `Dport` for attacks is 80.

(7) TotPkts, TotBytes, SrcBytes, Dur (original features)

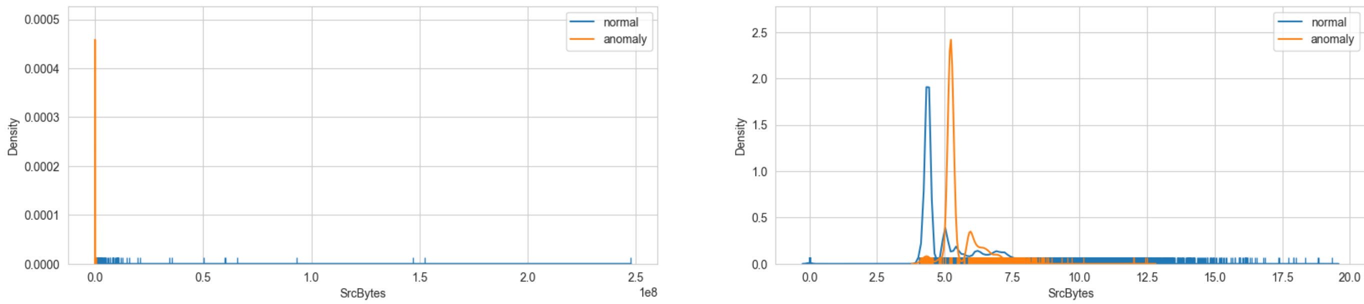
- Value ranges of these fields are large. Log scale can help visualising better.
- For 3 features `TotPkts, TotBytes, SrcBytes`, values of attack data are very close to 0, and the range is much narrower compared to normal data.



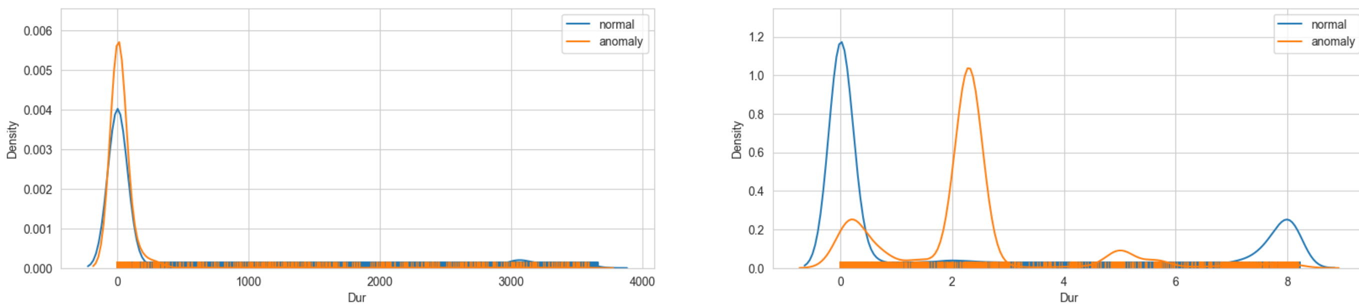
1) Visualisation of TotPkts



2) Visualisation of TotBytes



3) Visualisation of SrcBytes

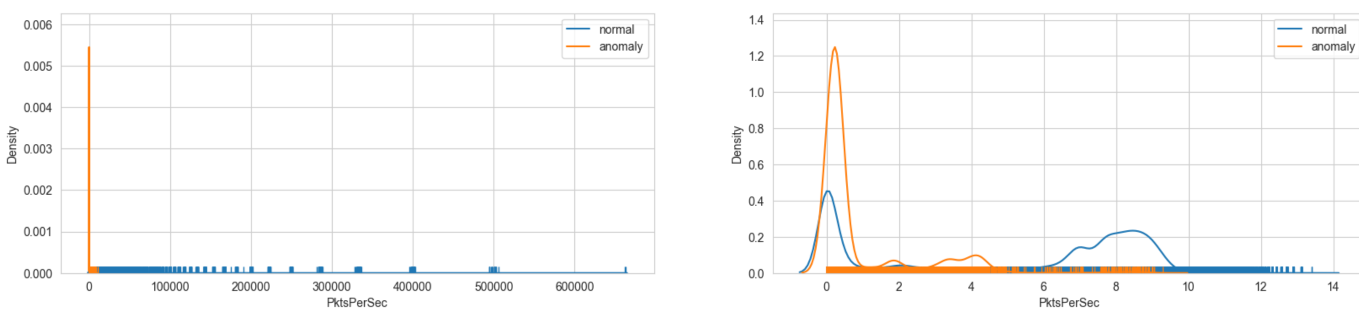


4) Visualisation of Dur

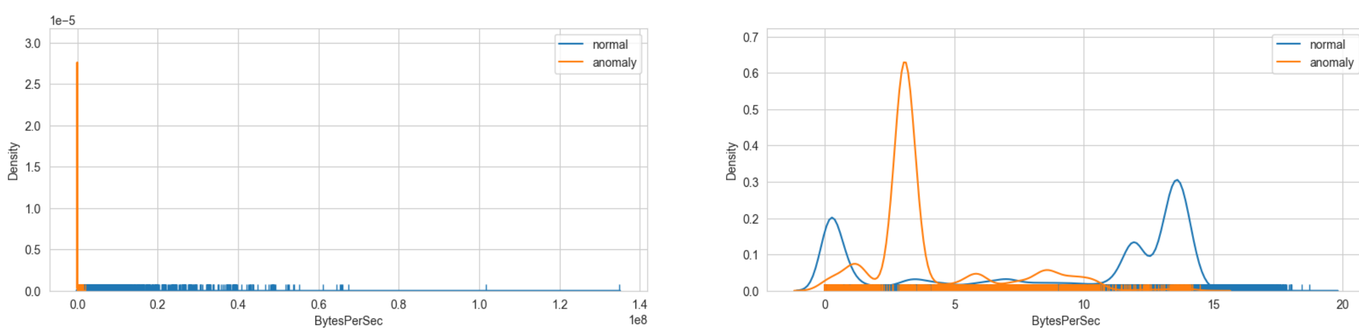
Fig I-18. Visualisation of TotPkts, TotBytes, SrcBytes, Dur

(8) PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt (calculated features)

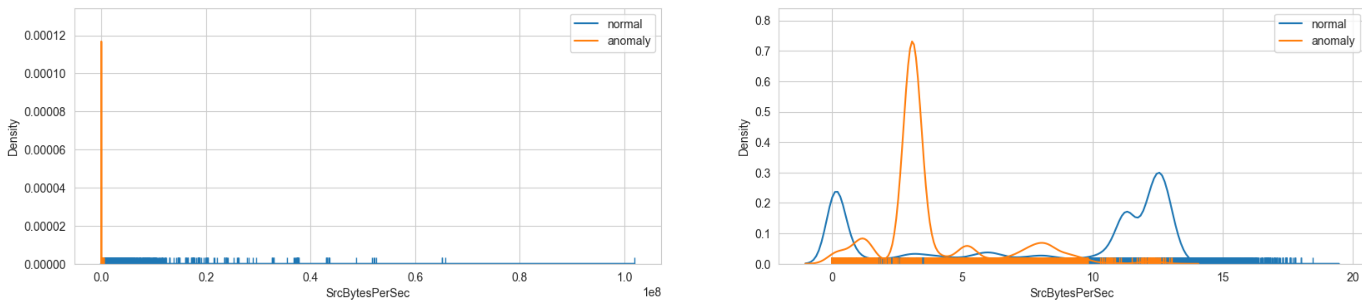
- Much higher peek in density of a value for attack data compared to normal data.



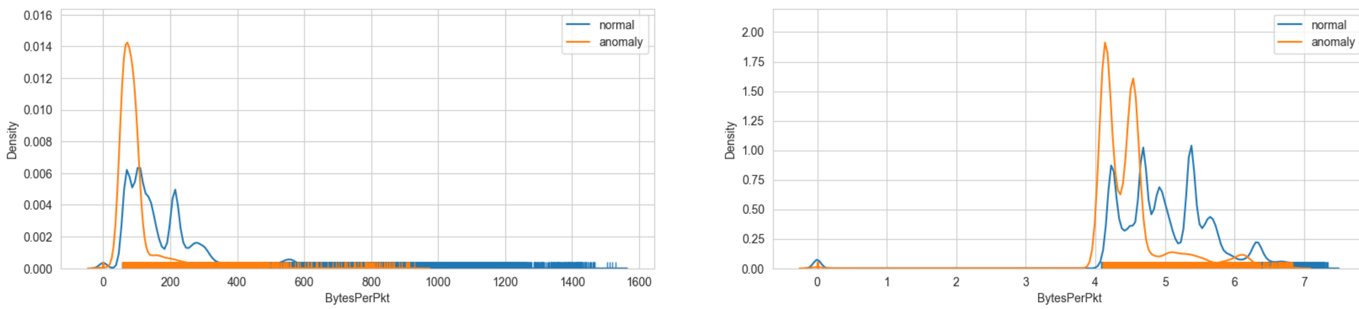
1) Visualisation of PktsPerSec



2) Visualisation of BytesPerSec



3) Visualisation of SrcBytesPerSec



4) Visualisation of BytesPerPkt

Fig I-19. Visualisation of PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt

3.3. Features selection

This section applies several techniques to rank the importance of features on the validation set.

3.3.1. Correlation

The code for this section is in Python, filename `vv2-1.ft_sel_1.ipynb`.

Getting the correlation matrix is a good way to know which features should be considered removed.

Applying pearson correlation on all features (including newly generated) results in a lot of correlation, which is understandable, as new features are generated based on original features.

(1) On original numeric features

In original numeric features, `TotPkts` and `TotBytes` have a high correlation.

	Dur	Sport	Dport	sTos	dTos	TotPkts	TotBytes	SrcBytes
Dur	1.000000	-0.226943	0.009626	0.019045	0.009231	0.001136	-0.002259	0.006228
Sport	-0.226943	1.000000	-0.008113	0.031876	0.285995	-0.017488	-0.018741	-0.005057
Dport	0.009626	-0.008113	1.000000	-0.000122	-0.010585	0.012782	0.009808	0.075802
sTos	0.019045	0.031876	-0.000122	1.000000	0.007286	-0.000204	-0.000126	0.000010
dTos	0.009231	0.285995	-0.010585	0.007286	1.000000	0.003538	0.003733	-0.002898
TotPkts	0.001136	-0.017488	0.012782	-0.000204	0.003538	1.000000	0.986756	0.272415
TotBytes	-0.002259	-0.018741	0.009808	-0.000126	0.003733	0.986756	1.000000	0.276162
SrcBytes	0.006228	-0.005057	0.075802	0.000010	-0.002898	0.272415	0.276162	1.000000

```
TotPkts          TotBytes          0.9867563448652203
```

Most correlated features are:

- TotPkts, TotBytes

Fig I-20. Correlation matrix on original features

When pair plotting these two features, we see as the `TotPkts` increase, `TotBytes` increase as well, and vice versa. The change is almost linear.

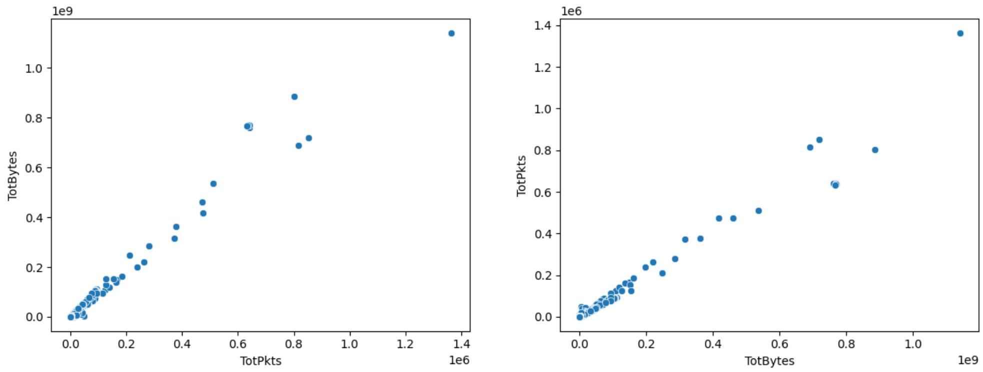


Fig I-21. Pair plot two features

(2) On calculated numeric features

	PktsPerSec	BytesPerSec	SrcBytesPerSec	BytesPerPkt
PktsPerSec	1.000000	0.898077	0.585165	-0.074692
BytesPerSec	0.898077	1.000000	0.686403	0.060703
SrcBytesPerSec	0.585165	0.686403	1.000000	-0.022180
BytesPerPkt	-0.074692	0.060703	-0.022180	1.000000

```
PktsPerSec       BytesPerSec       0.8980766382897152
PktsPerSec       SrcBytesPerSec    0.5851648292967798
BytesPerSec      SrcBytesPerSec    0.6864026841803242
```

Fig I-22. Correlation matrix on calculated numeric features

As `TotPkts` and `TotBytes` have high correlation, it is understandable that fields calculated based on these two fields would have high correlation as well, which are `PktsPerSec` and `BytesPerSec`. However, `SrcBytes` originally do not have high correlation with any features, yet its calculated feature `SrcBytesPerSec (=SrcBytes/Dur)` is now correlated with `PktsPerSec` and `BytesPerSec`.

3.3.2. Statistical hypothesis on numeric features

The code for this section is in Python, filename `vv2-1.ft_sel_1.ipynb`.

We conduct a simple statistical test on each numeric feature, with the assumption of two populations being approximately normally distributed, and test if the means and the standard deviations of the two populations are different. The procedure is as follows:

- We assume the samples are independent, and 2 populations are approximately normally distributed.
  - $Normal = X \sim N(\mu_1, \sigma_1^2)$
  - $Abnormal = Y \sim N(\mu_2, \sigma_2^2)$
- We conduct an f-test to check if 2 variances are equal.
  - Build hypothesis:

- $H_0' : \sigma_1^2 = \sigma_2^2$
- $H_1' : \sigma_1^2 \neq \sigma_2^2$

  - Calculate f_{score}
  - Calculate p_{value} (denoted as p_1)

- We conduct a 2-sample t-test to check if 2 means are qual.
  - Build hypothesis:
    - $H_0 : \mu_1 = \mu_2$
    - $H_1 : \mu_1 \neq \mu_2$
  - Calculate t_{score}
    - If 2 variances are equal $\sigma_1^2 = \sigma_2^2$

$$t_{score} = \frac{\overline{x_1} - \overline{x_2} - \Delta}{s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \quad where \quad s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2}}$$

    - If 2 variances are different $\sigma_1^2 \neq \sigma_2^2$ (H_0' is rejected)

$$t_{score} = \frac{\overline{x_1} - \overline{x_2} - \Delta}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \quad where \quad s_p = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}$$

  - Calculate p_{value} (denoted as p_2)

When analysing the result, we can see that for all features, p_1 is small enough to reject H_0', p_2 is small enough to reject H_0. This means we can draw a conclusion that if 2 populations are approximately normally distributed, they will have different means and standard deviations. These features therefore can be useful for detecting normal and attack flows or groups of flows. This method of analysis is meaningful, especially for Task II when we aggregate multiple flows into one record and use the mean value of aggregated flows for classification.

3.3.3. Chi2 contigency on categorical features

The code for this section is in Python, filename `vv2-1.ft_sel_1.ipynb`.

We conduct `chi2_contigency` test to examine the relationship between a categorical feature with the label (normal or abnormal), to see if they are independent or related to each other. This test of dependency between a feature with the label is to tell if the feature's values are distributed similarly across different values of label. For supervised learning models, the more significant dependency can imply that the feature is more meaningful as it represents that two variables share a similar distribution, and we might want to remove the features with low dependency. However, in unsupervised learning, the independency between the distribution of the feature and the label does not necessarily mean the feature is useless, as it indicates some values (of the feature) occur more in one class rather than the other. When analysing the result, we can see that for all categorical features, the dependency is insignificant.

3.3.4. Methods used in KDD competition

The code for this section is in Python, filename `vv2-2.ft_sel_2.ipynb`. The code is referenced heavily from https://github.com/solegalli/.

This approach is undertaken by data scientists at the University of Melbourne in the KDD 2009 data science competition [1]. This is a simple yet quite efficient method to contemplate the relationship between the feature and the label. The procedure is as follows:

For each categorical variable:

- Separate into train and test
- Compute the mean value of the target within each label of the categorical variable in the train set
- Use that mean target value per label as the prediction (in the test set) and calculate the ROC_AUC.

For each numerical variable:

- Separate into train and test
- Divide the variable into 100 quantiles
- Calculate the mean target within each quantile in the training set
- Use that mean target value / bin as the prediction (in the test set) and calculate the ROC_AUC.

The higher value of ROC_AUC indicates the more meaningful features. Running this method on the validation set gives the following result:

```
State       0.962036
Proto       0.917941
Dir         0.898655
Service     0.869783
dTos        0.822295
sTos        0.501622
dtype: float64
```

Fig I-23. Result on categorical features

```
Dur_binned                0.883219
Sport_binned              0.822717
SrcBytesPerSec_binned     0.811048
PktsPerSec_binned         0.805036
BytesPerSec_binned        0.771623
Dport_binned              0.737811
DstBytesPerSec_binned     0.539700
TotBytes_binned           0.536377
SrcBytes_binned           0.535586
BytesPerPkt_binned        0.505620
TotPkts_binned            0.505019
dTos_binned               0.499927
sTos_binned               0.499427
DstBytes_binned           0.302027
dtype: float64
```

Fig I-24. Result on numeric features

3.3.5. Mutual Information

The code for this section is in Python, filename `vv2-3.ft_sel_3.ipynb`.

(1) On all fields (fs1)

Running MI on all fields (including one-hot fields) results in the most important features as follow:

	Ft	Score
4	dTos	88859.653951
1	Sport	37461.567496
11	BytesPerPkt	3304.314082
0	Dur	2072.371795
9	BytesPerSec	1888.203135
13	DstBytesPerSec	1419.189073
8	PktsPerSec	1334.168626
10	SrcBytesPerSec	1038.303990
6	TotBytes	3.168939
12	DstBytes	3.043267
5	TotPkts	2.819844
2	Dport	1.474629
7	SrcBytes	0.412044
3	sTos	0.012200

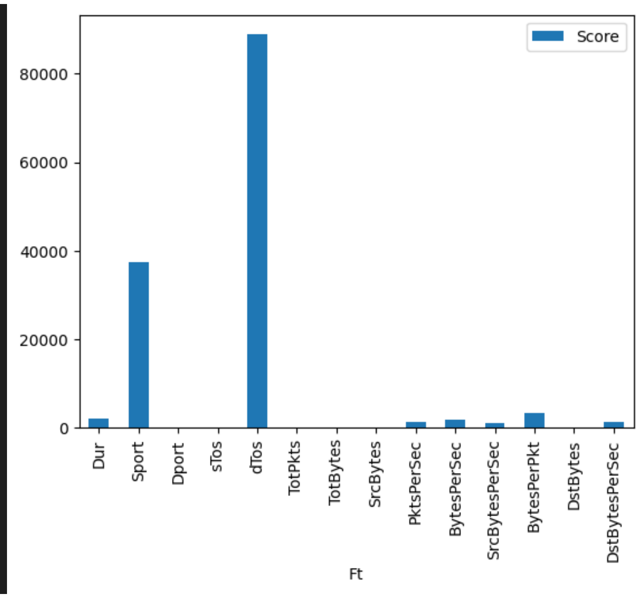


Fig I-25. Visualisation of importance scores for each feature

The features ranked highest are: `dTos, Sport, BytesPerPkt, Dur, BytesPerSec, DstBytesPerSec, PktsPerSec, SrcBytesPerSec`.

(2) Mutual information on numeric fields (fs2)

Since multiple the one-hot fields can represent only a string field. We will exclude these fields and apply the selection techniques on remaining numerical fields. The chosen features will be concatenated with the one-hot fields.

	Ft	Score
17	State_S_	600206.490796
4	dTos	88859.653951
39	Proto_tcp	69577.710759
38	Proto_udp	61868.289978
46	Service_80	60875.308540
52	Service_other	57804.403777
14	State_CON	49324.240597
50	Service_25	43529.128938
1	Sport	37461.567496
33	Flag_S	24338.048347
36	Flag_R	3814.990981
11	BytesPerPkt	3304.314082
0	Dur	2072.371795
51	Service_6667	2037.963675
9	BytesPerSec	1888.203135
15	State_alltcp	1705.498210
13	DstBytesPerSec	1419.189073
34	Flag_A	1363.198712
8	PktsPerSec	1334.168626
10	SrcBytesPerSec	1038.303990

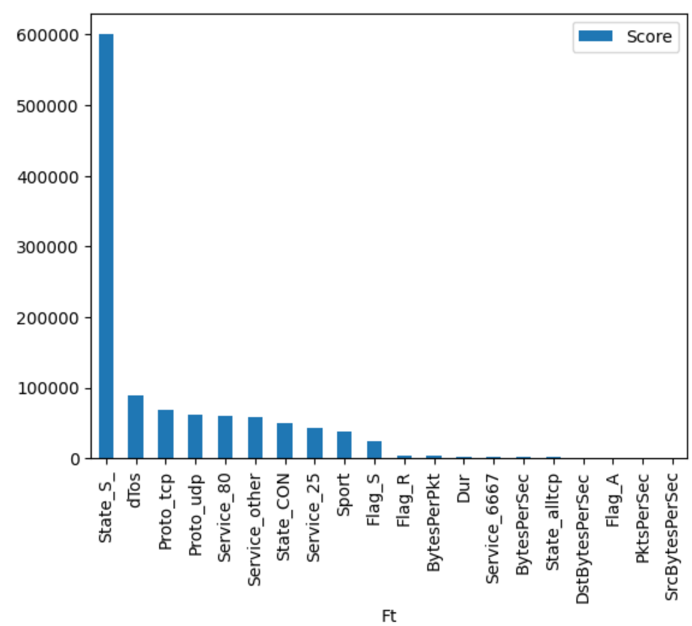


Fig I-26. Visualisation of importance scores for each feature

Top 20 features have the highest score are: `State_S_, dTos, Proto_tcp, Proto_udp, Service_80, Service_other, State_CON, Service_25, Sport, Flag_S, Flag_R, BytesPerPkt, Dur, Service_6667, BytesPerSec, State_alltcp, DstBytesPerSec, Flag_A, PktsPerSec, SrcBytesPerSec`.

3.3.6. Features selected for experiment

In this section, I describe 4 sets of features selected for experiments: `fs1, fs2, fs3, fs4`.

(1) fs1

After applying multiple techniques to analyse the features, we can see that categorical features are useful, within which 4 fields: `State` (after preprocessed as described in Section 2), `Proto`, `Dir`, `Service` have the highest relation score according to the method described in Section 3.3.4. Of all the values in these categorical fields, the values `State_S_, Proto_tcp, Proto_udp, Service_80, Service_other`, etc. (Fig I-26) are ranked highest. Because there can always be differences in the validation set that we use for features analysis and the train/test set, we should not explicitly choose only these fields. Instead, we should keep the one-hot fields of `State`, `Proto`, and `Service`. In case a categorical field has many values to be encoded, we can base on the scores of each one-hot value to determine which one-hot encoded fields to keep and which ones to drop. The one-hot encoded fields I choose for this feature set are: `State_CON, State_alltcp, State_INT, State_S_, State_URP, State_ECO, State_other, Flag_nan, Flag_S, Flag_A, Flag_P, Flag_R, Flag_F, Proto_udp, Proto_tcp, Proto_icmp, Proto_other, Service_80, Service_443, Service_21, Service_22, Service_25, Service_6667, Service_other`. It is noted that I do not use one-hot encoded fields for `Dir`, as the information about the flow's direction is implied in `State` and `Flag` fields. The numeric features I keep are: `sTos, Sport, Dport, TotPkts, TotBytes, SrcBytes, PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt`.

All features for `fs1` are:

```
sTos, Sport, Dport, TotPkts, TotBytes, SrcBytes, PktsPerSec, BytesPerSec,
SrcBytesPerSec, BytesPerPkt, State_CON, State_alltcp, State_INT, State_S_, State_URP,
State_ECO, State_other, Flag_nan, Flag_S, Flag_A, Flag_P, Flag_R, Flag_F, Proto_udp,
Proto_tcp, Proto_icmp, Proto_other, Service_80, Service_443, Service_21, Service_22,
Service_25, Service_6667, Service_other
```

(2) fs2

This set is composed of numeric features selected from Section 3.3.5(1) combined with all one-hot encoded features. The final set of features is:

```
dTos, Sport, BytesPerPkt, Dur, BytesPerSec, DstBytesPerSec, PktsPerSec, SrcBytesPerSec,
State_CON, State_alltcp, State_INT, State_S_, State_URP, State_ECO, State_RED,
State_REQ, State_ECR, State_URH, State_TXD, State_URFIL, State_R_, State_URN,
State_RSP, State_URHPRO, State_A_, State_other, Flag_nan, Flag_S, Flag_A, Flag_P,
Flag_R, Flag_F, Proto_udp, Proto_tcp, Proto_icmp, Proto_rtp, Proto_rtcp, Proto_igmp,
Proto_arp, Proto_other, Service_80, Service_443, Service_21, Service_22, Service_25,
Service_6667, Service_other
```

(3) fs3

This set is made up of features selected from Section 3.3.5(2), which are:

State_S_, dTos, Proto_tcp, Proto_udp, Service_80, Service_other, State_CON, Service_25, Sport, Flag_S, Flag_R, BytesPerPkt, Dur, Service_6667, BytesPerSec, State_alltcp, DstBytesPerSec, Flag_A, PktsPerSec, SrcBytesPerSec.

(4) fs4

This set consists of numeric features selected from Section 3.3.5(1) combined with 8 one-hot encoded fields selected from MI in Section 3.3.5(2):

dTos, Sport, BytesPerPkt, Dur, BytesPerSec, DstBytesPerSec, PktsPerSec, SrcBytesPerSec, State_S_, Proto_tcp, Proto_udp, Service_80, Service_other, State_CON, Service_25, Flag_S

This is to compare the models' performance on `fs1` (do not explicitly choose selected one-hot encoded fields) and `fs4` (explicitly choose 8 selected one-hot encoded fields).

4. Anomaly detection

This section conducts 14 experiments of 2 models on 7 sets of features.

4.1. Experiment setups

7 experiment setups are as follow:

- Exp01: Using numeric features in original data.
- Exp02: Using numeric features.
- Exp03: Using all features (including original and generated).
- Exp04: Using fs1.
- Exp05: Using fs2.
- Exp06: Using fs3.
- Exp07: Using fs4.

Table I-4. Experimentation setups description

	Number of ft	Features
Exp01	8	Dur, sTos, dTos, Sport, Dport, TotPkts, TotBytes, SrcBytes
Exp02	14	Dur, sTos, dTos, Sport, Dport, TotPkts, TotBytes, SrcBytes, PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt, DstBytes, DstBytesPerSec
Exp03	53	Dur, sTos, dTos, Sport, Dport, TotPkts, TotBytes, SrcBytes, PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt, DstBytes, DstBytesPerSec, State_CON, State_alltcp, State_INT, State_S_, State_URP, State_ECO, State_RED, State_REQ, State_ECR, State_URH, State_TXD, State_URFIL, State_R_, State_URN, State_RSP, State_URHPRO, State_A_, State_other, Flag_nan, Flag_S, Flag_A, Flag_P, Flag_R, Flag_F, Proto_udp, Proto_tcp, Proto_icmp, Proto_rtp, Proto_rtcp, Proto_igmp, Proto_arp, Proto_other, Service_80, Service_443, Service_21, Service_22, Service_25, Service_6667, Service_other
Exp04	34	dTos, Sport, Dport, TotPkts, TotBytes, SrcBytes, PktsPerSec, BytesPerSec, SrcBytesPerSec, BytesPerPkt, State_CON, State_alltcp, State_INT, State_S_, State_URP, State_ECO, State_other, Flag_nan, Flag_S, Flag_A, Flag_P, Flag_R, Flag_F, Proto_udp, Proto_tcp,

		`Proto_icmp, Proto_other, Service_80, Service_443, Service_21, Service_22, Service_25, Service_6667, Service_other`
Exp05	47	`dTos, Sport, BytesPerPkt, Dur, BytesPerSec, DstBytesPerSec, PktsPerSec, SrcBytesPerSec, State_CON, State_alltcp, State_INT, State_S_, State_URP, State_ECO, State_RED, State_REQ, State_ECR, State_URH, State_TXD, State_URFIL, State_R_, State_URN, State_RSP, State_URHPRO, State_A_, State_other, Flag_nan, Flag_S, Flag_A, Flag_P, Flag_R, Flag_F, Proto_udp, Proto_tcp, Proto_icmp, Proto_rtp, Proto_rtcp, Proto_igmp, Proto_arp, Proto_other, Service_80, Service_443, Service_21, Service_22, Service_25, Service_6667, Service_other`
Exp06	20	`State_S_, dTos, Proto_tcp, Proto_udp, Service_80, Service_other, State_CON, Service_25, Sport, Flag_S, Flag_R, BytesPerPkt, Dur, Service_6667, BytesPerSec, State_alltcp, DstBytesPerSec, Flag_A, PktsPerSec, SrcBytesPerSec`
Exp07	16	`dTos, Sport, BytesPerPkt, Dur, BytesPerSec, DstBytesPerSec, PktsPerSec, SrcBytesPerSec, State_S_, Proto_tcp, Proto_udp, Service_80, Service_other, State_CON, Service_25, Flag_S`

4.2. Evaluation metrics

For anomaly detection, it is important to balance between precision and recall of abnormal class and the accuracy of the model. The time of the detected attack can also be important. For example, in practical application, it is acceptable to have lower recall score with higher precision if the model can spot a number of early anomaly traffic to block the IP in time.

However, it is out of the scope of this report to discuss the evaluation metrics for such scenarios. This report will discuss the result and the flows detected as bots in Section 4, but for evaluation, I aim for higher recall score for minority class (anomaly) while keeping the ROC_AUC of the model high (expectedly higher than 75%).

Output of outlier detection has -1 value to indicate outliers. For easier evaluation using the scikit-learn evaluation metrics functions, I convert the output of the model to 0 and 1, where 0 indicates normal flows, and 1 indicates outliers.

The summary result of the 2 models is shown in the Table I-5 and Table I-6 below.

Table I-5. iForest results on 7 experiments

iForest	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07
Training time	**27.7s**	27.4s	46.4s	36.2s	42.6s	31.9s	31.1s
Testing time	9.46s	10.1s	16.2s	10.6s	11.6s	9.8s	**9.16s**
Recall (class 1)	9.28	15.43	59.32	**65.02**	58.95	51.99	53.58
Precision (class 1)	2.90	5.31	14.17	**17.36**	15.48	14.55	15.87
f1 (class 0)	99.22	99.30	99.22	99.33	99.29	99.31	**99.35**
f1 (class 1)	4.42	7.90	22.88	**27.40**	24.53	22.74	24.49
AUC	54.04	57.18	78.96	**81.91**	78.85	75.40	76.24
Accuracy	98.45	98.61	98.45	98.67	98.59	98.63	**98.72**

Table I-6. LOF (as novelty detection model) results on 7 experiments

LOF	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07
Training time	**3m16s**	4m16s	34m52s	24m36s	34m26s	21m39s	28m23s
Testing time	**1m39s**	2m12s	17m39s	13m30s	14m	13m42s	12m20s
Recall (class 1)	3.44	10.13	58.17	42.51	**58.68**	49.73	49.73
Precision (class 1)	1.16	2.49	10.60	7.55	10.88	10.36	**11.11**
f1 (class 0)	**99.24**	99.05	98.96	98.87	98.98	99.06	99.12
f1 (class 1)	1.74	3.99	17.93	12.82	**18.35**	17.14	18.16
AUC	51.14	54.29	78.13	70.24	**78.40**	74.03	74.09
Accuracy	**98.49**	98.11	97.94	97.76	97.98	98.14	98.26

4.3. iForest

The algorithm attempts to separate one observation from others by splitting the data points. It randomly selects a feature and then randomly select a split value between the maximum and minimum values of the selected feature. Since the anomalies often don't cluster together, an anomaly can be isolated in a few steps, while a normal observation may require more steps to be separated.

4.3.1. iForest and parameters

The tuning code is in Python, filename `vv4.iForest.__1__.ipynb`.

The parameters for iForest model (using scikit learn) are `n_estimators`, `contamination`. After multiple experiments, the chosen parameters are `n_estimators = 35, contamination = 0.01`. It is noted that these parameters are tuned while experimenting on the validation set only. The validation set is split into train and test set with the ratio of 70:30.

4.3.2. Experiments and results

The code for this section is in Python, filename `vv5.iforest.__1__.ipynb`.

The detailed result for each experiment is demonstrated in the source code file. Table I-7 and Table I-8 show the detail of the best result, which is of Exp04:

Table I-7. Confusion matrix

752410	9170
1036	1926

Table I-8. Classification matrix

	precision	recall	f1-score	score
0	0.9986	0.9880	0.9933	761580
1	0.1736	0.6502	0.2740	2962
Accuracy			0.9867	764542
Macro avg	0.5861	0.8191	0.6336	764542
Weighted avg	0.9954	0.9867	0.9905	764542
AUC				0.8191

4.3.3. Scores

The code for this section is in Python, filename `vv6.after_predict.iforest.ipynb`.

Analysing the score of detected records, we see that true bots are scored with lower negative values:

```
-0.311085    3428
-0.311441    3149
-0.311626    3110
-0.316132    3054
-0.312745    2897
             ...
-0.340814       1
-0.415229       1
-0.324897       1
-0.340646       1
-0.414406       1
Name: Score_iForest, Length: 124441, dtype: int64

count    752410.000000
mean         -0.339238
std           0.032884
min          -0.464701
25%          -0.356757
50%          -0.323950
75%          -0.313865
max          -0.310354
Name: Score_iForest, dtype: float64
```

True normal scores

```
-0.653373    41
-0.651970    37
-0.648786    36
-0.635999    33
-0.655341    30
             ..
-0.606535     1
-0.638273     1
-0.465388     1
-0.609119     1
-0.485490     1
Name: Score_iForest, Length: 598, dtype: int64

count    1926.000000
mean       -0.615474
std         0.047183
min        -0.718684
25%        -0.644749
50%        -0.622359
75%        -0.603962
max        -0.464727
Name: Score_iForest, dtype: float64
```

True bot scores

Fig I-27. Scores by iForest

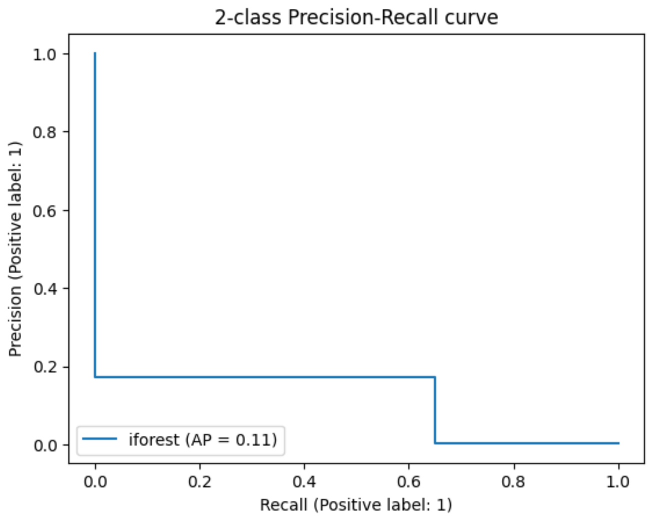
Using the threshold of 2% outliers, we get the threshold score is `-0.44918331`. If we set threshold percentiles to 2(%), or the threshold score to `-0.44918331`, only records that are scored lower than this threshold score (`-0.44918331`) is classified as bot.

Applying the threshold technique with different threshold percentiles, we get the results illustrated in Table I-9 and Fig I-28.

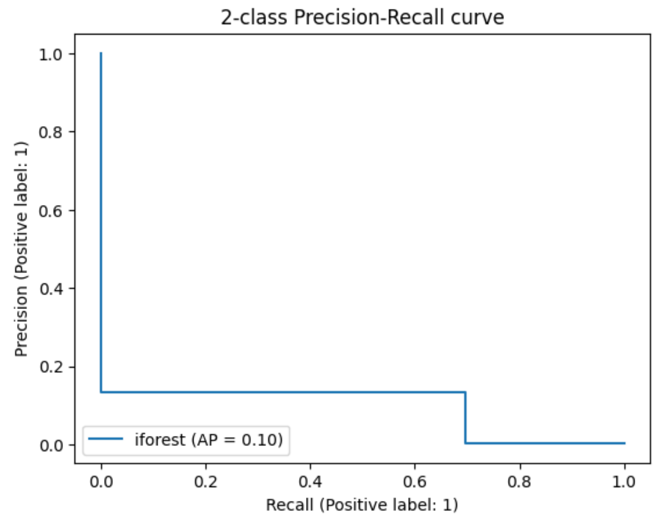
Table I-9. Result when applying different thresholds

thresh_p	thresh_score	Precision (class 1)	Recall (class 1)	f1 (class 0)	f1 (class 1)	Accuracy	AUC
-	-	17.36	65.02	99.33	27.40	98.67	81.91
2	-0.44918331	13.51	69.75	99.07	22.64	98.15	84.01
1	-0.47863293	24.05	62.09	99.54	34.67	99.09	80.66
0.5	-0.53062718	46.27	59.72	99.79	52.14	99.58	79.73
0.45	-0.54221799	51.24	59.52	99.81	55.07	99.62	79.65
0.4	-0.55450336	56.98	58.85	99.83	57.90	99.67	79.34

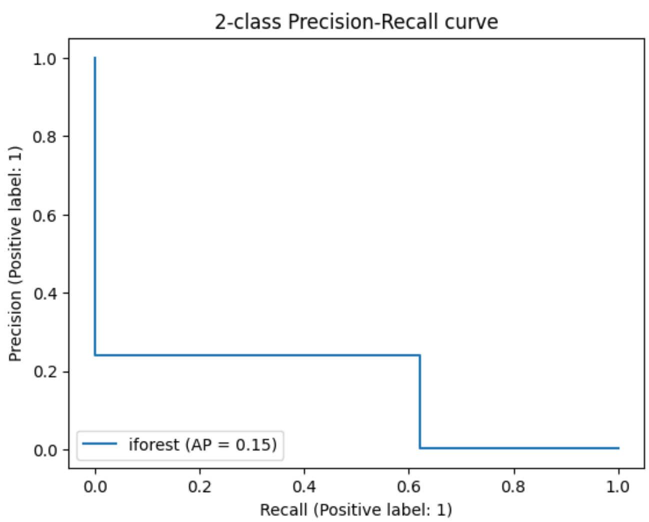
Because bots are scored with lower negative values, lower threshold gives higher precision, however, in trade-off of lower recall.



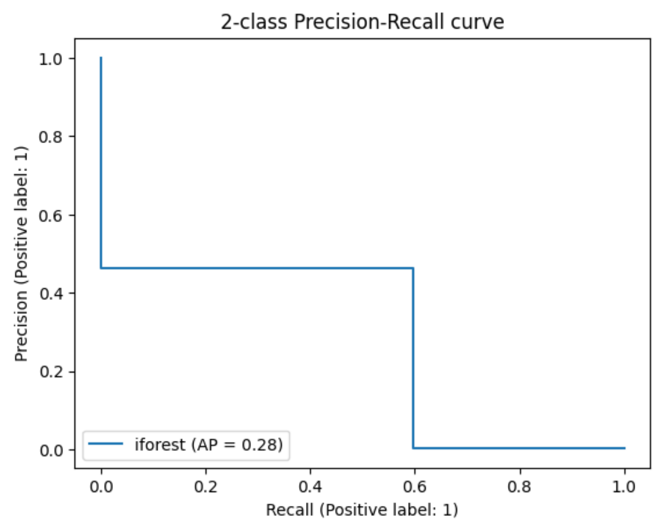
Non-threshold (model.predict() output)



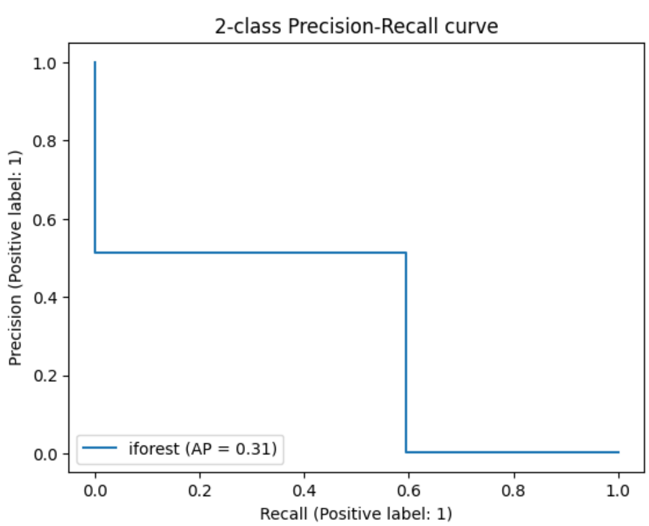
p=2 (2% outliers)



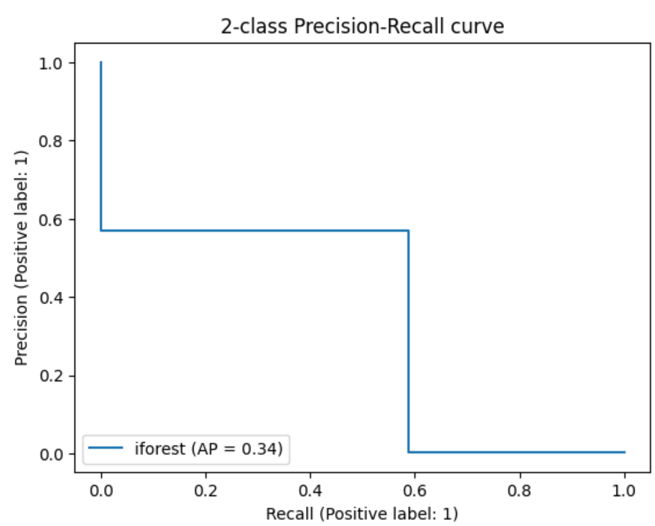
p=1 (1% outliers)



p=0.5 (0.5% outliers)



p=0.45 (0.45% outliers)



p=0.4 (0.4% outliers)

Fig I-28. ROC curve on different thresholds

4.3.4. Post-processing and simple thresholding

The code that applies this technique is in `vv6.after_predict.iforest.ipynb`.

In most machine learning tasks, pre-processing and post-processing are quite important, as some simple processing can reduce noise and improve the result quite significantly.

For an attack to cause impacts on the server (for example, an attack that performs a CKC (Cyber Kill Chain)), it needs to conduct a series of actions, which means the model should be able to detect multiple flows coming from a true bot as anomalies. In other words, flows detected as bots will be more likely to be bots if they come from the same `SrcAddr`, as opposed to when only few flows from a `SrcAddr` are reported as bots, which might be noise. We can do that by considering the ratio of number of flows detected as bots over the total flows from one `SrcAddr` (`p=n_flows_detected/n_flows_total`), as well as the number of flows detected as bots from one `SrcAddr` (`n_flows_detected`).

It should be noted that this tactic might not work well with DDoS attack detection, and this method of post-processing will require analysing the detected batch after a certain time period (annotated as `T_post`). Therefore, it will decrease the capability to detect early anomalies if `T_post` is too large. On the contrary, if `T_post` is too small, this technique might be useless. It is necessary to choose appropriate `T_post` and threshold values.

In this experiment, I did not select `T_post`, I simply apply on the whole set. Table I-10 illustrates the result after applying this technique with respect to each `thresh_p`. The default threshold I chose for post-processing technique is `num_thresh=100` and `per_thresh=0.5`. num_thresh and per_thresh should be chosen based on analysing data knowing the labels for the optimal use, which makes this technique not precisely unsupervised technique. I did not tune these parameters based on validation set (or any set), instead I just chose random values for `num_thresh` and `per_thresh`.

- `num_thresh`: Only `SrcAddr` that has more than `num_thresh` flows detected as bot is classified as bot (`n_flows_detected > num_thresh`), otherwise consider them as normal.
- `per_thresh`: Only `SrcAddr` that has `n_flows_detected/n_flows_total > per_thresh` detected as bot is classified as bot, otherwise consider them as normal.

Table I-10. Comparison when not applying and applying post-processing technique

thresh_p		No post-processing	Post-processing
-	**Precision (class 1)**	17.36	67.46
	Recall (class 1)	65.02	65.02
	f1 (class 0)	99.33	99.87
	f1 (class 1)	27.40	66.22
	Accuracy	98.67	99.74
	AUC	81.91	82.45
2	**Precision (class 1)**	13.51	52.25
	Recall (class 1)	69.75	69.75
	f1 (class 0)	99.07	99.82

	f1 (class 1)	22.64	59.75
	Accuracy	98.15	99.64
	AUC	84.01	84.74
1	**Precision (class 1)**	24.05	76.02
	Recall (class 1)	62.09	62.09
	f1 (class 0)	99.54	99.89
	f1 (class 1)	34.67	68.35
	Accuracy	99.09	99.78
	AUC	80.66	81.01
0.5	**Precision (class 1)**	46.27	90.39
	Recall (class 1)	59.72	59.72
	f1 (class 0)	99.79	99.91
	f1 (class 1)	52.14	71.93
	Accuracy	99.58	99.82
	AUC	79.73	79.85
0.45	**Precision (class 1)**	51.24	91.63
	Recall (class 1)	59.52	59.52
	f1 (class 0)	99.81	99.91
	f1 (class 1)	55.07	72.17
	Accuracy	99.62	99.82
	AUC	79.65	79.75
0.4	**Precision (class 1)**	56.98	93.16
	Recall (class 1)	58.85	58.85
	f1 (class 0)	99.83	99.91
	f1 (class 1)	57.90	72.13
	Accuracy	99.67	99.82
	AUC	79.34	79.41

As predicted, this post-processing technique is quite helpful in cases where recall (class 1) is high and precision (class 1) is low (equivalent to high `thresh_p`) as this technique is to reduce false alarms, and has less impact when `thresh_p` is low.

4.4. LOF

The Local Outlier Factor (LOF) algorithm is density-based anomaly detection technique. It calculates the local density deviation of a particular data point with respect to its neighbours. The samples that have a significantly lower density than their neighbours are regarded as outliers.

4.4.1. LOF and parameters

The tuning code is in Python, filename `vv4.lof.__1__.ipynb`.

The parameters for LOF model (using scikit learn) are `n_neighbors`, `contamination`. Running LOF takes quite much more time compared to iForest, it is difficult to conduct multiple experiments. The chosen parameters are `n_neighbors = 35, contamination = 0.01`. It is noted that these parameters are tuned while experimenting on the validation set only. The validation set is split into train and test set with the ratio of 70:30.

4.4.2. LOF as outlier detection and novelty detection

The code for this section is in `vv5.lof.__1__.ipynb` and `vv5.lof.testonly.__1__.ipynb`.

Because LOF can be used as both outlier detection and novelty detection (scikit-learn documentation). These are two approaches for anomaly detection, where in outlier detection, the training data contains outliers which are defined as observations that are far from the others, while in novelty detection, the training data is not polluted by outliers and the method detects whether new observations are outliers. Scikit-learn's implementation of LOF as outlier detection does not allow train and test on different sets, but on one dataset only with `fit_predict()` method. On the contrary, the implementation of LOF as novelty detection has `fit()` method to fit on train set (expectedly not containing any anomalous samples), and `predict()` method to predict outliers in test set.

I have experimented using LOF in both ways: as outlier detection and novelty detection. The results show that using LOF as outlier detection (`fit_predict()` on test set only) gives worse results than when LOF is used as novelty detection (`fit()` on train set and `predict()` on test set). The code and detailed experiment results for LOF as outlier detection are in `vv5.lof.__1__.testonly.ipynb`. , for LOF as novelty detection are in `vv5.lof.__1__.ipynb`. Table I-11 shows the comparison results of LOF being used in two ways. It is noted that for the best use of novelty detection, the train set should not contain any anomaly. Because the requirement of this task is to not use the Label in the train set, the results shown in this section are of training with the train data containing both normal and abnormal samples.

Table I-11. Results of LOF when used as outlier detection and novelty detection

	Exp01		**Exp02**		**Exo03**		**Exp04**		**Exp05**		**Exp06**		**Exp07**	
	OD	ND	OD	ND	OD	ND	OD	ND	OD	ND	OD	ND	OD	ND
f1 (class 0)	99.32	99.24	99.32	99.05	99.31	98.96	99.32	98.87	99.31	98.98	99.31	99.06	99.31	99.12
f1 (class 1)	2.49	1.74	2.09	3.99	1.38	17.93	2.09	12.82	1.87	18.35	1.81	17.14	1.40	18.16
AUC	51.73	51.15	51.38	54.29	50.74	78.13	51.38	70.24	51.18	78.40	51.12	74.03	50.75	74.09
Accuracy	98.65	98.49	98.64	98.11	98.63	97.94	98.64	97.76	98.64	97.98	98.64	98.14	98.63	98.26

It can be seen from Table I-11 that when using LOF as outlier detection, there is not much change among different feature sets. On the contrary, when LOF is used as novelty detection, the accuracy varies quite significantly depending on feature sets.

4.4.3. Experiments and results

The code for this section is in Python, filename `vv5.lof.__1__.ipynb`.

The detailed result for each experiment is demonstrated in the source code file. Table I-12**Error! Reference source not found.** and Table I-13**Error! Reference source not found.** show the detail of the best result, which is of Exp04:

Table I-12. Confusion matrix

747340	14240
1224	1738

Table I-13. Classification matrix

	precision	recall	f1-score	score
0	0.9984	0.9813	0.9898	761580
1	0.1088	0.5868	0.1835	2962
Accuracy			0.9798	764542
Macro avg	0.5536	0.7840	0.5866	764542
Weighted avg	0.9949	0.9798	0.9866	764542
AUC				0.7840

4.4.4. Scores

The code for this section is in Python, filename `vv6.after_predict.lof.ipynb`.

Analysing the score of detected records, we see that true bots are scored with lower negative values:

```
-1.000000    11708
-1.724696      327
-1.940665      318
-2.059889       74
-0.926494       39
             ...
-0.992878        1
-1.610417        1
-0.980114        1
-0.987052        1
-1.131971        1
Name: Score_lof, Length: 727248, dtype: int64

count    747340.000000
mean         -1.113110
std           0.206653
min          -2.645851
25%          -1.134327
50%          -1.036004
75%          -1.002608
max          -0.094757
Name: Score_lof, dtype: float64
```

True normal scores

```
-2.889343e+03    8
-2.897020e+03    4
-1.075152e+05    3
-8.170037e+06    3
-6.795430e+00    3
                ..
-1.272918e+01    1
-1.839161e+01    1
-1.311334e+01    1
-1.022639e+01    1
-6.196954e+01    1
Name: Score_lof, Length: 1683, dtype: int64

count    1.738000e+03
mean    -4.737092e+05
std      2.696559e+06
min     -4.654950e+07
25%     -2.870912e+01
50%     -1.518613e+01
75%     -5.429338e+00
max     -2.646778e+00
Name: Score_lof, dtype: float64
```

True bot scores

Fig I-29. Scores by LOF

Table I-14 illustrated results when applying the threshold technique with different threshold percentiles.

Table I-14. Result when applying different thresholds

thresh_p	thresh_score	Precision (class 1)	Recall (class 1)	f1 (class 0)	f1 (class 1)	Accuracy	AUC
-	-	10.88	58.68	98.98	18.35	97.98	78.40

2	-2.73695291	11.31	58.41	99.02	18.96	98.07	78.31
1	-5.26878301	17.32	44.70	99.48	24.96	98.96	71.93
0.5	-15.53062718	22.21	28.66	99.67	25.03	99.33	64.14
0.45	-20.02304846	20.95	24.34	99.67	22.52	99.35	61.99
0.4	-25.11533590	18.01	18.60	99.68	18.30	99.36	59.14

4.4.5. Post-processing and simple thresholding

The code that applies this technique is in `vv6.after_predict.lof.ipynb`.

I used the same `num_thresh` and `per_thresh` as in iForest experiment (section 4.3.4) for this experiment. Table I-15 illustrates the result after applying this technique with respect to each `thresh_p`.

Table I-15. Comparison when not applying and applying post-processing technique

thresh_p		No post-processing	Post-processing
-	**Precision (class 1)**	10.88	51.65
	Recall (class 1)	58.68	58.68
	f1 (class 0)	98.98	99.81
	f1 (class 1)	18.35	54.94
	Accuracy	97.98	99.63
	AUC	78.40	79.23
2	**Precision (class 1)**	11.31	53.71
	Recall (class 1)	58.41	58.41
	f1 (class 0)	99.02	99.82
	f1 (class 1)	18.96	55.96
	Accuracy	98.07	99.64
	AUC	78.31	79.11
1	**Precision (class 1)**	17.32	0
	Recall (class 1)	44.70	0
	f1 (class 0)	99.48	99.75
	f1 (class 1)	24.96	0
	Accuracy	98.96	99.51
	AUC	71.93	49.95

The same behaviour as demonstrated in Section 4.3.4 is observed. However, for LOF, when `thresh_p=1`, the model can only identify 1324/2962 flows as bots (recall = 0.45), which is why using `per_thresh=0.5` will result in all detected flows are ignored.

5. Discussion

Correctly detected flows are saved at `vv6.__1__.iforest.exp04_play.__81.91__.true_bots.csv` and `vv6.__1__.lof.exp05_mi1.__78.4__.true_bots.csv`.

It can be seen that ICMP and TCP scanning flows (SYN scan) are the easiest type of attacks to detect using anomaly detection.

```
flow=From-Botnet-V45-TCP-Attempt-SPAM                      980
flow=From-Botnet-V45-ICMP                                  812
flow=From-Botnet-V46-TCP-Not-Encrypted-SMTP-Private-Proxy-1    104
flow=From-Botnet-V46-TCP-Attempt                            21
flow=From-Botnet-V46-TCP-Attempt-SPAM                        6
flow=From-Botnet-V45-TCP-CC106-IRC-Not-Encrypted             2
flow=From-Botnet-V46-TCP-CC1-HTTP-Not-Encrypted              1
Name: LabelStr, dtype: int64
```

Fig I-30. Types of attacks correctly identified as anomalies

Stream ID	Start Time	Proto	State	Dport	Label Strnbsp;	Label	Label_iforest
316722	2021-08-12 22:0	icmp	other	0	flow=From-Botnet-V45-ICMP	5	1
316726	2021-08-12 22:0	icmp	other	0	flow=From-Botnet-V45-ICMP	5	1
316737	2021-08-12 22:0	icmp	other	0	flow=From-Botnet-V45-ICMP	5	1
751902	2021-08-13 01:1	tcp	S_	25	flow=From-Botnet-V46-TCP-Attempt-SPAM	5	1
762849	2021-08-13 01:1	tcp	S_	25	flow=From-Botnet-V46-TCP-Attempt-SPAM	5	1
701458	2021-08-13 01:0	tcp	alltcp	6667	flow=From-Botnet-V46-TCP-CC1-HTTP-Not-Encrypted	5	1
702499	2021-08-13 01:0	tcp	alltcp	65498	flow=From-Botnet-V46-TCP-Not-Encrypted-SMTP-Private-Proxy-1	5	1
703244	2021-08-13 01:0	tcp	alltcp	65501	flow=From-Botnet-V46-TCP-Not-Encrypted-SMTP-Private-Proxy-1	5	1
703507	2021-08-13 01:0	tcp	alltcp	65500	flow=From-Botnet-V46-TCP-Not-Encrypted-SMTP-Private-Proxy-1	5	1

Fig I-31. Attack flows identified as anomalies

However, DNS requests, web establish requests, or binary download from bots are difficult to be detected. This is understandable, as binary download action is difficult to be acknowledged as malware downloading without knowing the payload. It seems that the model cannot detect C2C communication as well (`TCP-CC1-HTTP-Not-Encrypted, TCP-WEB-Established, TCP-CC5-Plain-HTTP-Encrypted-Data, TCP-Established-Custom-Encryption-1`, etc.).

```
flow=From-Botnet-V46-TCP-Established-Custom-Encryption-1        17
flow=From-Botnet-V46-TCP-HTTP-Google-Net-Established-2          17
flow=From-Botnet-V46-TCP-Established-SSL-To-Microsoft-6         16
flow=From-Botnet-V46-TCP-Established-HTTP-Ad-46                 13
flow=From-Botnet-V46-TCP-Established-SSL-To-Microsoft-7         13
flow=From-Botnet-V46-TCP-Attempt-SPAM                           13
flow=From-Botnet-V46-TCP-CC12-HTTP-Not-Encrypted                11
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-7        10
flow=From-Botnet-V46-TCP-WEB-Established-SSL                    10
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-4         9
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-5         9
flow=From-Botnet-V46-TCP-CC5-Plain-HTTP-Encrypted-Data           7
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-Live-1    7
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-Live-3    7
flow=From-Botnet-V46-TCP-Established-SSL-To-Microsoft-4          6
flow=From-Botnet-V46-TCP-Established-HTTP-To-Microsoft-Live-2    6
flow=From-Botnet-V46-TCP-WEB-Established                         5
flow=From-Botnet-V46-TCP-Established-SSL-To-Microsoft-5          5
flow=From-Botnet-V45-TCP-HTTP-Google-Net-Established-6           4
flow=From-Botnet-V46-TCP-CC1-HTTP-Not-Encrypted                  4
flow=From-Botnet-V46-TCP-Established-HTTP-Binary-Download-3      4
```

Fig I-32. Types of attacks missed

516164	2021-08-12 22:4	tcp	alltcp	5680	flow=From-Botnet-V45-TCP-CC73-Not-Encrypted	5	0
516181	2021-08-12 22:4	tcp	alltcp	5677	flow=From-Botnet-V45-TCP-CC73-Not-Encrypted	5	0
517268	2021-08-12 22:5	udp	CON	53	flow=From-Botnet-V45-UDP-DNS	5	0
517628	2021-08-12 22:5	tcp	alltcp	80	flow=From-Botnet-V45-TCP-Established-HTTP-Ad-40	5	0
680640	2021-08-13 00:5	tcp	alltcp	80	flow=From-Botnet-V46-TCP-HTTP-Google-Net-Established-6	5	0
683768	2021-08-13 00:5	udp	CON	53	flow=From-Botnet-V46-UDP-DNS	5	0
683777	2021-08-13 00:5	tcp	alltcp	80	flow=From-Botnet-V46-TCP-WEB-Established	5	0
684077	2021-08-13 00:5	udp	CON	53	flow=From-Botnet-V46-UDP-DNS	5	0
684119	2021-08-13 00:5	tcp	alltcp	80	flow=From-Botnet-V46-TCP-Established-HTTP-Binary-Download-1	5	0
684620	2021-08-13 00:5	tcp	alltcp	80	flow=From-Botnet-V46-TCP-Established-HTTP-Binary-Download-1	5	0

Fig I-33. Examples of Attack flows missed

Detecting C2C communication is a challenging task. It is extremely difficult to distinguish such communications from normal flows as the behaviours at this stage is “less automatic” compared to DDoS or scanning attacks, especially if the C2 server uses trusted certificates. Even if the C2 server uses self-signed certificates, we can only mark such flows as warning. For HTTP plain request, we can analyse the payload to see if any malicious content is transferred. However, if the HTTP payload is encrypted by the attacker’s encryption implementation, it can make payload-based detectors suffer. There is a need to combine multiple techniques to prevent intrusions in time, including traffic header/payload analysis, and files’ behaviours monitoring.

6. References

[1] H. Miller *et al.*, “Predicting customer behaviour: The University of Melbourne’s KDD Cup report,” in *Proceedings of KDD-Cup 2009 Competition*, New York, New York, USA, Jun. 2009, vol. 7, pp. 45–55. [Online]. Available: https://proceedings.mlr.press/v7/miller09.html

Task II

1. Introduction

Task II of this report demonstrates how to use supervised models for network traffic classification (normal and attack) and discuss how to bypass such a model. Task II is structured as follow: Section 2 describes how to aggregate flows by Pattern into one aggregated record, uses feature selection techniques to select the best set of features, Section 3 demonstrates the use of a simple model (LogisticRegression) in classifying attack records, Section 4 shows the describes the application of a simple attack method (FGSM) to generate adversarial samples to bypass the model, Section 5 illustrates the process of reproducing the attack flows so that the detection pipeline is completely fooled.

2. Aggregating flows

The features for each flow are generated in the same method as in Task I. However, in Task II, the aim is to detect malicious IP but not traffic flow, one method could be applying the model on flows data and choosing the IP with more than a threshold of flows detected as bot (for example, an IP having more than 50% of flows being classified as bot would be considered as bot IP). This method is quite costly, and the fundamental goal is different. A bot could perform an action not necessarily considered abnormal but follow a pattern. For example, a bot will connect to the C&C server to issue command or sending data. These activities usually happen in interval. In fact, when analysing the train and validation set, we can see repeated patterns every 30 second (in train set) and (in test set)

	Start Time	Conversation	Proto	Label Str
138565	2022-07-27 05:31:57.733069	150.35.87.168 -> 127.149.226.168	tcp	flow=From-Botnet-V44-TCP-Attempt
138566	2022-07-27 05:31:57.788240	150.35.87.168 -> 127.149.226.169	tcp	flow=From-Botnet-V44-TCP-Attempt
138567	2022-07-27 05:31:57.798263	150.35.87.168 -> 127.149.226.170	tcp	flow=From-Botnet-V44-TCP-Attempt
138568	2022-07-27 05:31:57.808271	150.35.87.168 -> 127.149.226.171	tcp	flow=From-Botnet-V44-TCP-Attempt
138569	2022-07-27 05:31:57.818338	150.35.87.168 -> 127.149.226.172	tcp	flow=From-Botnet-V44-TCP-Attempt
138570	2022-07-27 05:31:57.828368	150.35.87.168 -> 127.149.226.173	tcp	flow=From-Botnet-V44-TCP-Attempt
138571	2022-07-27 05:31:57.838301	150.35.87.168 -> 127.149.226.174	tcp	flow=From-Botnet-V44-TCP-Attempt
138572	2022-07-27 05:31:57.848433	150.35.87.168 -> 127.149.226.175	tcp	flow=From-Botnet-V44-TCP-Attempt
138573	2022-07-27 05:31:57.858442	150.35.87.168 -> 127.149.226.176	tcp	flow=From-Botnet-V44-TCP-Attempt
138574	2022-07-27 05:31:57.868519	150.35.87.168 -> 127.149.226.177	tcp	flow=From-Botnet-V44-TCP-Attempt
138754	2022-07-27 05:32:18.607911	150.35.87.168 -> 127.149.226.178	tcp	flow=From-Botnet-V44-TCP-Attempt
138755	2022-07-27 05:32:18.708620	150.35.87.168 -> 127.149.226.179	tcp	flow=From-Botnet-V44-TCP-Attempt
138756	2022-07-27 05:32:18.708767	150.35.87.168 -> 127.149.226.180	tcp	flow=From-Botnet-V44-TCP-Attempt
138757	2022-07-27 05:32:18.708772	150.35.87.168 -> 127.149.226.181	tcp	flow=From-Botnet-V44-TCP-Attempt
138758	2022-07-27 05:32:18.708918	150.35.87.168 -> 127.149.226.182	tcp	flow=From-Botnet-V44-TCP-Attempt
138759	2022-07-27 05:32:18.708928	150.35.87.168 -> 127.149.226.183	tcp	flow=From-Botnet-V44-TCP-Attempt
138760	2022-07-27 05:32:18.709001	150.35.87.168 -> 127.149.226.184	tcp	flow=From-Botnet-V44-TCP-Attempt
138761	2022-07-27 05:32:18.709074	150.35.87.168 -> 127.149.226.185	tcp	flow=From-Botnet-V44-TCP-Attempt
138762	2022-07-27 05:32:18.709201	150.35.87.168 -> 127.149.226.186	tcp	flow=From-Botnet-V44-TCP-Attempt
138763	2022-07-27 05:32:18.709276	150.35.87.168 -> 127.149.226.187	tcp	flow=From-Botnet-V44-TCP-Attempt
139072	2022-07-27 05:32:39.638557	150.35.87.168 -> 127.149.226.188	tcp	flow=From-Botnet-V44-TCP-Attempt
139073	2022-07-27 05:32:39.738546	150.35.87.168 -> 127.149.226.189	tcp	flow=From-Botnet-V44-TCP-Attempt
139074	2022-07-27 05:32:39.738638	150.35.87.168 -> 127.149.226.190	tcp	flow=From-Botnet-V44-TCP-Attempt
139075	2022-07-27 05:32:39.738776	150.35.87.168 -> 127.149.226.191	tcp	flow=From-Botnet-V44-TCP-Attempt
139076	2022-07-27 05:32:39.738790	150.35.87.168 -> 127.149.226.192	tcp	flow=From-Botnet-V44-TCP-Attempt
139077	2022-07-27 05:32:39.738893	150.35.87.168 -> 127.149.226.193	tcp	flow=From-Botnet-V44-TCP-Attempt
139078	2022-07-27 05:32:39.738909	150.35.87.168 -> 127.149.226.194	tcp	flow=From-Botnet-V44-TCP-Attempt
139079	2022-07-27 05:32:39.739050	150.35.87.168 -> 127.149.226.195	tcp	flow=From-Botnet-V44-TCP-Attempt
139080	2022-07-27 05:32:39.739234	150.35.87.168 -> 127.149.226.196	tcp	flow=From-Botnet-V44-TCP-Attempt

Fig II-1. Bot records in train set

	Start Time	Conversation	Proto	Label Str
40549	2022-07-25 23:58:17.171311	150.35.87.168 -> 84.169.249.201	icmp	flow=From-Botnet-V44-ICMP
107427	2022-07-26 01:01:52.902156	150.35.87.168 -> 150.35.83.12	udp	flow=From-Botnet-V44-UDP-DNS
109653	2022-07-26 01:04:13.488181	150.35.87.168 -> 150.35.83.12	udp	flow=From-Botnet-V44-UDP-DNS
109658	2022-07-26 01:04:13.610498	150.35.87.168 -> 77.210.4.21	tcp	flow=From-Botnet-V44-TCP-WEB-Established
109664	2022-07-26 01:04:13.963712	150.35.87.168 -> 150.35.83.12	udp	flow=From-Botnet-V44-UDP-DNS
109665	2022-07-26 01:04:14.065630	150.35.87.168 -> 77.210.4.21	tcp	flow=From-Botnet-V44-TCP-WEB-Established
258676	2022-07-26 04:32:40.178019	150.35.87.168 -> 122.60.75.29	icmp	flow=From-Botnet-V44-ICMP
531153	2022-07-26 10:27:52.522377	150.35.87.168 -> 205.106.55.150	icmp	flow=From-Botnet-V44-ICMP
941614	2022-07-26 18:44:48.824258	150.35.87.168 -> 90.7.206.69	icmp	flow=From-Botnet-V44-ICMP
1106325	2022-07-26 23:09:22.481392	150.35.87.168 -> 150.35.83.12	udp	flow=From-Botnet-V44-UDP-DNS
1106362	2022-07-26 23:09:26.980818	150.35.87.168 -> 150.35.83.12	udp	flow=From-Botnet-V44-UDP-DNS
1106363	2022-07-26 23:09:26.984072	150.35.87.168 -> 98.103.251.27	tcp	flow=From-Botnet-V44-TCP-WEB-Established

Fig II-2. Bot records in validation set

To detect attack patterns, we can check the mean value of aggregated flows. If these mean values are similar, it might indicate repeated behaviour. The idea of aggregating flows by `SrcAddr` using time window is introduced in the work of [2]. In this report, I aggregate by `Conversation`, `Proto` and `State`. If aggregating only by `SrcAddr`, it might be more complicated to represent the Proto and State values of all the flows been grouped together.

The same technique of generating and encoding features in task I is used for this task. The feature engineering step is conducted on train and validation set. After the analysis, selected features are: `BytesPerPkt_mean, PktsPerSec_mean, BytesPerSec_mean, Sport_max, Sport_mean, n_flows, BytesPerPkt_max, BytesPerSec_max, SrcBytesPerSec_max, P_tcp, P_udp, P_other, S_CON, S_alltcp, S_INT, S_RED, S_other, S_ECO`

The selected values for time `window-width` and `window-stride` are 7200 and 3600 seconds (2-hour-width and 1-hour-stride), respectively.

It is noted that because the aim of the model in this task is to detect attack patterns (repeated behaviours), and given that we choose a wide time window, we assume within the selected time window, a bot will repeat its behaviour at least once. Therefore, all the aggregated records whose `total_flows = 1` will be dropped. This significantly reduces the number of records in our dataset. This action takes the risk of omitting repeated behaviour occurring within more than 2 hours. Therefore, it should be carefully done after analysing the captured data and with appropriate time window width.

3. Detecting bot IP using supervised learning

Because we choose to aggregate by `Conversation`, the model will classify a conversation rather than an IP. A simple `LogisticRegression` model is used, with `class_weight` parameter set to `balanced`, to assign a suitable weight for each class to handle imbalanced data.

The train and validation set are merged and split with ratio 70:30 into `train_split` and `test_split` for training and evaluation.

The model's result is demonstrated in Table II-1. (class 1 is bot, class 0 is normal)

Table II-1. Model result on Train and Test set

	train_split	**test_split**	**test**
AUC	95.35	95.29	97.39
Accuracy	90.69	90.58	94.80
Recall (class 1)	100.00	100.00	100.00

4. Attack the model using FGSM

There are two types of adversarial attack: One is untargeted attack, where we only aim for the model to not classify the sample as true label. The other is targeted attack, where we aim for the model to classify the sample as one specific class. In this task we need to fool the model into misclassifying the true bots as normal (specific class). Since the model used is binary classifier, performing an untargeted attack will help us achieve the goal of targeting attack as well.

In this report I used a simple method to generate adversarial samples which is Fast Gradient Sign Method (FGSM). FGSM aims to add noise to the data to trick the model into misclassifying an input. To do that, the noise should have the same direction as the gradient of the cost function with respect to the data. This is to increase the loss of the model's prediction of the true label (bot); hence it will result in predicting the tampered input as the other label (normal).

To constrain the perturbation that the noise creates to the original data, an epsilon value is used to scale the noise. Epsilon should be small to create as little change in data as possible.

The formula of FGSM is as follows:

$$x_{adv} = x + \epsilon * sgn(\nabla_x L(\theta, x, y))$$

where x is the original input we would like to perturb to fool the model. x_{adv} is the perturbed sample from x, ϵ is a small value we use to control the noise, y is the true label of x, $L(\theta, x, y)$ is the loss function of model θ with input x.

4.1. Choose an IP to perturb features

The code for this section is in `u4-6.ipynb`.

To select a sample from the model output, we analyse the records predicted as bots.

First, a simple threshold is applied, we keep only those records whose bot score (score for class 1) is greater than 0.75 (high confidence). The model result is increased as follow.

Table II-2. Original result of model output on test set

	precision	recall	f1-score	Score
0	1.0000	0.9479	0.9732	18166
1	0.0596	1.0000	0.1125	60
Accuracy			0.9480	18226
Macro avg	0.5298	0.9739	0.5429	18226
Weighted avg	0.9969	0.9480	0.9704	18226
AUC				0.9739

Table II-3. New result on test set after applying scoring threshold

	precision	recall	f1-score	Score
0	1.0000	0.9620	0.9806	18166
1	0.0799	1.0000	0.1480	60
Accuracy			0.9621	18226
Macro avg	0.5399	0.9810	0.5643	18226
Weighted avg	0.9970	0.9621	0.9779	18226
AUC				0.9810

Then for each conversation, we compute how many aggregated records are marked as bots.

	n_agg_total	n_agg_detected	n_flows_total	n_flows_detected	p_agg
41.232.73.23 -> 150.35.87.168	60	60	120	120	1.000000
150.35.87.62 -> 189.216.27.38	5	5	28	28	1.000000
150.35.87.62 -> 101.204.145.180	4	4	10	10	1.000000
150.35.87.62 -> 210.49.127.175	4	4	10	10	1.000000
150.35.87.62 -> 44.135.158.222	4	4	19	19	1.000000
...	...	...	...	...	...
150.35.87.174 -> 72.168.158.73	15	1	200	2	0.066667
150.35.87.62 -> 77.128.235.216	15	1	498	2	0.066667
150.35.87.62 -> 209.3.175.130	44	2	88	4	0.045455
150.35.87.174 -> 150.35.83.12	69	2	65130	4	0.028986
150.35.87.174 -> 88.229.62.197	53	1	914	2	0.018868

361 rows × 5 columns

Fig II-3. Statistics of detected records by each conversation

`p_agg` is computed as `n_agg_detected / n_agg_total`.

We can see from the table above conversation `41.232.73.23 -> 150.35.87.168` has `p_agg=1` (all records are marked as bot behaviours), and also, the number of aggregated records being labelled as malicious is much higher than the others. 2 IPs `41.232.73.23 (SrcAddr)` and `150.35.87.168 (DstAddr)` are also the only 2 IPs whose all the records are marked as bot.

Therefore, conversations between these two IPs are chosen to generate adversarial samples.

4.2. Generate adversarial samples

The code to generate adversarial samples is in `vv4-7.adversarial.ipynb`. The steps are:

- Choose an IP address that we want to trick the model into misclassifying.
- Select records to compute adversarial samples.
- Compute direction matrix.
- Compute adversarial samples with chosen epsilon values.
- Retrieve the model's output.

Table II-4. Adversarial attack results with respect to different epsilon

epsilon	Number of perturbed samples successfully exploit the model	
0.000020	3/60	5.00%
0.000021	6/60	10.00%
0.000022	59/60	98.33%
0.000023	60/60	100.00%

With `epsilon=0.000023`, all perturbed samples can successfully fool the model.

However, this is the values after being processed. The following section will demonstrate how to reproduce raw values and attack flows based on the computed processed values.

5. Reproduce the attack flows

5.1. Modify/Generate flows

The code that demonstrates the process of reproducing attack flows is in `u4-7.adversarial.ipynb`.

In the security domain, the data are preprocessed into numeric features. Hence, after getting an adversarial sample that can fool the model, it can be challenging to reproduce flows that can bypass the model after being preprocessed, yet still be able to retain the malicious functionality.

We select an aggregated record to demonstrate how to reproduce an attack flow that can bypass the model. The aggregated records having a smaller value of `n_flows` can suggest easier reproduction. Because all the records have the same `n_flows` value of 2 (Fig II-4), we randomly select a record (Fig II-5).

	SrcAddr	DstAddr	State	Proto	n_flows	StreamID_unique	Sport_nunique	Sport_mean	Label	window_id
2218	41.232.73.23	150.35.87.168	alltcp	tcp	2	[1976 51591]	2	6668.0	5	0
3615	41.232.73.23	150.35.87.168	alltcp	tcp	2	[51591 84191]	2	6666.5	5	1
4322	41.232.73.23	150.35.87.168	alltcp	tcp	2	[84191 101058]	2	6667.0	5	2
4675	41.232.73.23	150.35.87.168	alltcp	tcp	2	[101058 109906]	2	6667.5	5	3
4777	41.232.73.23	150.35.87.168	alltcp	tcp	2	[109906 115675]	2	6667.5	5	4
4933	41.232.73.23	150.35.87.168	alltcp	tcp	2	[115675 121315]	2	6667.0	5	5
5080	41.232.73.23	150.35.87.168	alltcp	tcp	2	[121315 127423]	2	6667.5	5	6
5164	41.232.73.23	150.35.87.168	alltcp	tcp	2	[127423 132401]	2	6667.0	5	7
5243	41.232.73.23	150.35.87.168	alltcp	tcp	2	[132401 179483]	2	6665.5	5	8
5310	41.232.73.23	150.35.87.168	alltcp	tcp	2	[179483 191231]	2	6665.5	5	9
5373	41.232.73.23	150.35.87.168	alltcp	tcp	2	[191231 198879]	2	6667.0	5	10
5437	41.232.73.23	150.35.87.168	alltcp	tcp	2	[198879 206653]	2	6668.5	5	11
5491	41.232.73.23	150.35.87.168	alltcp	tcp	2	[206653 214274]	2	6667.0	5	12
5549	41.232.73.23	150.35.87.168	alltcp	tcp	2	[214274 235458]	2	6667.0	5	13
5610	41.232.73.23	150.35.87.168	alltcp	tcp	2	[235458 242887]	2	6668.5	5	14
5675	41.232.73.23	150.35.87.168	alltcp	tcp	2	[242887 250728]	2	6668.5	5	15
5750	41.232.73.23	150.35.87.168	alltcp	tcp	2	[250728 258696]	1	6668.0	5	16
5829	41.232.73.23	150.35.87.168	alltcp	tcp	2	[258696 266623]	1	6668.0	5	17
5898	41.232.73.23	150.35.87.168	alltcp	tcp	2	[266623 274907]	1	6668.0	5	18
5976	41.232.73.23	150.35.87.168	alltcp	tcp	2	[274907 283187]	2	6666.5	5	19
6067	41.232.73.23	150.35.87.168	alltcp	tcp	2	[283187 291371]	2	6667.0	5	20
6164	41.232.73.23	150.35.87.168	alltcp	tcp	2	[291371 299722]	2	6668.5	5	21
6278	41.232.73.23	150.35.87.168	alltcp	tcp	2	[299722 309916]	2	6666.0	5	22
6382	41.232.73.23	150.35.87.168	alltcp	tcp	2	[309916 318812]	2	6665.0	5	23
6585	41.232.73.23	150.35.87.168	alltcp	tcp	2	[318812 329788]	2	6666.5	5	24
6813	41.232.73.23	150.35.87.168	alltcp	tcp	2	[329788 339180]	2	6666.5	5	25
6994	41.232.73.23	150.35.87.168	alltcp	tcp	2	[339180 356440]	2	6667.5	5	26
7217	41.232.73.23	150.35.87.168	alltcp	tcp	2	[356440 366439]	2	6666.5	5	27

Fig II-4. All selected aggregated records have the same `n_flows`

	index	Conversation	SrcAddr	DstAddr	State	Proto	n_flows	StreamID_unique	Sport_nunique	Sport_mean	...	P_udp	P_other	S_CON	S_alltcp	S_INT	S_RED	S_other	S_ECO	Label	Label_Pred
0	2218	41.232.73.23 -> 150.35.87.168	41.232.73.23	150.35.87.168	alltcp	tcp	2	[1976 51591]	2	6668.0	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1	1

1 rows × 92 columns

Fig II-5. Selected aggregated record

Two flows being aggregated into this record are:

	StreamID	StartTime	Dur	Proto	SrcAddr	Sport	Dir	DstAddr	Dport	State	...	SrcBytes	Label	LabelStr	PktsPerSec	BytesPerSec	SrcBytesPerSec	BytesPerPkt	DstBytes	DstBytesPerSec	State_orig
1975	1976	2022-07-25 23:30:50.093002	3571.229975	tcp	41.232.73.23	6669	<?>	150.35.87.168	1027	alltcp	...	3750	5	flow=From-Botnet-V44-TCP-CC107-IRC-Not-Encrypted	0.022401	1.648452	1.050058	73.587500	2137	0.598393	PA_PA
51590	51591	2022-07-26 00:32:56.114638	3527.809143	tcp	41.232.73.23	6667	<?>	150.35.87.168	1027	alltcp	...	9319	5	flow=From-Botnet-V44-TCP-CC107-IRC-Not-Encrypted	0.062078	5.494345	2.641583	88.506849	10064	2.852762	PA_PA

2 rows × 24 columns

Fig II-6. Selected flows

As the model takes in 18 features: `BytesPerPkt_mean, PktsPerSec_mean, BytesPerSec_mean, Sport_max, Sport_mean, n_flows, BytesPerPkt_max, BytesPerSec_max, SrcBytesPerSec_max, P_tcp, P_udp, P_other, S_CON, S_alltcp, S_INT, S_RED, S_other, S_ECO`

where `P_tcp, P_udp, P_other, S_CON, S_alltcp, S_INT, S_RED, S_other, S_ECO` are generated from categorical features `Proto` and `State`. Assume these values are critical for a network flow (or packet) to function correctly, we will retain values for these fields. When we analyse two flows being aggregated into one record, we see the `Sport` value looks like a service port, which should not be changed as well.

The value of `n_flows` can be changed but can only be increased (to ensure the bot intention is retained).

After choosing to retain 11 features: `Sport_max, Sport_mean, P_tcp, P_udp, P_other, S_CON, S_alltcp, S_INT, S_RED, S_other, S_ECO`, the minimum value for `epsilon` so that this record can bypass the model is `0.00005`.

With `epsilon=0.00005`, perturbed values for 6 changeable features are:

Table II-5. Computed processed values and changes with `epsilon=0.00005`

	Processed value		Values for aggregated record	
Features	**New value**	**Changes**	**New value**	**Old value**
BytesPerPkt_mean	5.28652304e-02	-5.e-05	80.970593	81.047175
PktsPerSec_mean	5.00845864e-05	5.e-05	25.010623	0.04224
BytesPerSec_mean	5.00180146e-05	5.e-05	9916.089546	3.571398
BytesPerPkt_max	5.65317162e-02	5.e-05	88.585199	88.506849
BytesPerSec_max	5.00186306e-05	5.e-05	14750.948178	5.494345
SrcBytesPerSec_max	5.00089598e-05	5.e-05	14743.93356	2.641583
n_flows	5e-05	5.e-05		2

The *new value for aggregated record* column indicates the minimum value (if the change is > 0) or maximum value (if the change is < 0) that the new aggregated record must have to exploit the model `(1)`

The original value of `n_flows` is small (`= 2`), and is encoded into `0` using `MinmaxScaler`; the new processed value for processed `n_flows` is `5e-5` is larger than the original encoded value. Therefore, we will not compute the new value for `n_flows`; instead, from `(1)`, we assume changes in `n_flows` (that either result in changing the processed value or not) would not affect the result.

We can infer from Table II-5 that in order to reproduce the attack, the `BytesPerPkt_mean` value needs to decrease, while `BytesPerPkt_max` needs to increase, in order to retrieve this, we will need to increase the

BytesPerPkt value of one flow from Fig II-6, and add another flow with value smaller than current BytesPerPkt_min to reduce the mean value.

We can see from Table II-5 that 2 features witness the changes in 2 aggregated values, mean and max, which are BytesPerSec and BytesPerPkt. In order to perform changes in two aggregated values of 1 feature, we might need to either add 2 flows or add 1 flow and modify the value of an existing flow. Since we need to manipulate 4 aggregated values, we need to insert a minimum of 2 flows. We, therefore, will use these 2 flows to manipulate these features without touching these features of the original flows. We notice that for other features, the changes are quite large, we will use the inserted flows to perturb these values as well.

The final flows needed in order should be:

Table II-6. Calculated features values for inserted flows to change aggregated values

	BytesPerPkt	PktsPerSec	BytesPerSec	SrcBytesPerSec
1976	73.587500	0.022401	1.648452	1.050058
51591	88.506849	0.062078	5.494345	2.641583
#1 (change max)	88.585199	49.97901	14750.948178	14743.93356
#2 (change mean)	73.202824	49.97901	24906.267209	
Aggregated values need to change				
Max	**88.585199**		**14750.948178**	**14743.93356**
Mean	**80.970593**	**25.010623**	**9916.089546**	

We see that in order to change the mean of BytesPerSec to 9916.0895, we would need to change the value of BytesPerSec for the #2 inserted flow to 24906.2672, which is larger than the max value (14750), but it should not be a problem as the BytesPerSec_max value of the aggregated record will then increase (to 24906.2672), which will lead to the change in the same intended direction after it is processed.

Combining with other fields, we retrieve 4 flows as follows to reproduce the attack:

Table II-7. Raw values for 4 flows within a window

	BytesPerPkt	PktsPerSec	BytesPerSec	SrcBytesPerSec	Sport	Proto	State
1976	73.587500	0.022401	1.648452	1.050058	6667	tcp	PA_PA
51591	88.506849	0.062078	5.494345	2.641583	6669	tcp	PA_PA
#1	88.585199	49.97901	14750.948178	14750.948178	6667	tcp	PA_PA
#2	73.202824	49.97901	24906.267209	24906.267209	6669	tcp	PA_PA

Proto and State values must remain the same (so that 4 flows would be aggregated together). As noticed, the value of State is PA_PA, which means it needs the response from the DstAddr. Because of this reason, we will fill 6667 and 6669 in Sport fields for two inserted flows to remain the mean and max value as well as to ensure the service port is open for the flows to have two directions (if we fill 6668 instead, the mean and max will not change but there is no guarantee that the service port is open, and there will be less chances that the flow would have the same State value).

Because we are aiming to fool the model into misclassifying the first aggregated record, we arrange the inserted flows in a way that it will appear in the first window, but not in the second one. The final flows (for the first window) are stored in `result/dfo_new1.csv`.

5.2. Test the model performance on new flows

The code to test the model performance on new flows is in `u4-8.test-attack.ipynb`.

We feed the flows to the pipeline of the program to test the model performance, we can see how the model has been tricked into misclassification.

```
""" check output of the model """
y_pred = model.predict(X)
print('y true', y)
print('y pred', y_pred)
[114]  ✓ 0.3s                                   Python
...  y true [1]
     y pred [0]
```

Fig II-7. Model misclassified the attack flows as expected

6. Discussion

The example in this section demonstrates how to reproduce the attack flows to trick the model.

However, this report only reproduces the `[…]PerSec` fields, but not the original `TotPkts`, `TotBytes`, `SrcBytes` and `Dur` fields. It is difficult to reproduce both values of `TotPkts` and `Dur` without assuming one value. We can analyse the average `Dur` value and craft the packets to change `TotPkts` and `SrcBytes` accordingly. To craft the `SrcBytes`, we must also know in advance the `DstBytes`, which could be calculated by sending a request to the server to retrieve default response. In other words, `DstBytes` should be based on the response of the `DstAddr`.

Moreover, it can be seen that generated flows have to have high `SrcBytesPerSec`. (at least one flow has to achieve `SrcBytesPerSec=14750.948178`). However, to achieve such a high value for `SrcBytesPerSec` is very difficult, as it depends quite significantly on maximum packet size allowed (configured by the network manager), bandwidth size, and network speed.

Furthermore, even though we can compute the value for the flows, it is still a challenging task to reproduce each packet, especially if we do not know how the flows are generated from captured packets.

In conclusion, reproducing a network attack which can bypass a supervised learning model is challenging even though the method to generate adversarial samples is quite straightforward, due to the way the data is preprocessed before feeding the model.

7. References

[2] K. Xu, Z.-L. Zhang, and S. Bhattacharyya, "Reducing Unwanted Traffic in a Backbone Network.," *SRUTI*, vol. 5, pp. 9–15, 2005.