A UNSW-NB15 machine learning project

Wilfred Van Casteren
Open Universiteit
Masteropleiding AI

December 12, 2023

Abstract

In this article I will show the some of the basic steps a machine learning project needs to go through. I used Python and its machine learning libraries to read the data, inspect, preprocess and test the dataset. A decision tree classifier was used to test the and retest the datamodel. A decision tree classifier, a support vector machine based classifier and a k nearest neighbor were compared using the datamodel. The dataset used for this purpose, is aimed at Intrusion Detection Systems (IDSs), so I will discuss IDSs briefly after covering the programming environment I used.

Contents

1 Getting started **3**
1.1 Preliminaries . . . 3
1.2 Intrusion Detection Systems(IDS) and the UNSW-NB15 dataset . . . 3
1.2.1 IDS . . . 3
1.2.2 The UNSW-NB15 dataset . . . 4
1.2.3 Read, inspect and clean the data . . . 4
1.2.4 First impression of the data . . . 5

2 Inspection and preprocessing **6**
2.1 Numerical features . . . 6
2.1.1 Basic features . . . 6
2.1.2 Content features . . . 10
2.1.3 Time features . . . 13
2.1.4 Additional generated features . . . 17
2.1.5 Connection features . . . 18
2.1.6 Code to inspect numerical feature set . . . 21
2.2 Handling numerical features . . . 21
2.3 Handling categorical features . . . 22
2.3.1 Grouping categorical values . . . 22
2.3.2 Categorical values as binary features . . . 26
2.4 Handling features with low variance . . . 26
2.5 Features included in dataset . . . 26

3 Zero measuring, adapting and retesting the brand new model **27**
3.1 Zero measurement . . . 27
3.1.1 Results . . . 27
3.1.2 Code . . . 28
3.2 Highly correlated features . . . 29
3.3 Retesting . . . 30
3.3.1 Performance change after correlation reduction . . . 31
3.4 Further reducing features using Lasso . . . 32
3.4.1 Pairplots . . . 32
3.4.2 Reduce features using Lasso . . . 33
3.5 Retesting after Lasso reduction . . . 34
3.6 Performance change after a partial Lasso reduction . . . 34

4 Comparing the model on different classifiers **36**
4.1 Training and testing several different classifiers . . . 36
4.1.1 Results . . . 36
4.1.2 Thoughts on hyperparameters . . . 37
4.1.3 Code . . . 37

Conclusion **41**

Future work 42

A Extras 45

Chapter 1

Getting started

1.1 Preliminaries

There is a multitude of ways to work with Python and its libraries, I prefer **PyCharm**, a dedicated Python IDE. If needed it handles configuration, version management, inline documentation,... out of the box. Poetry is my favorite dependency management tool. It really is a blessing to have PyCharm by your side.

This report describes a software project using Python packages like pandas, matplotlib, scikit-learn, numpy, seaborn, wakepy, scipy. For things like file and directory handling some other packages were used too. A common way to refer to numpy, seaborn, pandas and seaborn packages is by using respectively 'np', 'sns' and 'pd', so if in the code in this report these 'namespaces' are used, they are referring to these Python packages. Furthermore needs to be remarked that not every bit of code written will be included in this report, but the most important pieces are.

- pandas: a Python package providing expressive data structures, also see pandas in 10 minutes
- scikit-learn: a machine learning package, the commonly used namespace for it being sklearn
- numpy: n-dimensional arrays, dependency for pandas, scikit-learn,...
- matplotlib: library for creating visualisations
- seaborn: visualisations library based on matplotlib
- wakepy: library to prevent your computer from sleeping
- scipy: a library for fundamental scientific algorithms

1.2 Intrusion Detection Systems(IDS) and the UNSW-NB15 dataset

1.2.1 IDS

The goal of an IDS is to identify different kinds of malicious network traffic and computer usage, which cannot be identified by a traditional firewall. This is vital to achieving high protection against actions that compromise the availability, integrity, or confidentiality of computer systems [1].

An intrusion can be defined as any kind of unauthorised activities that cause damage to an information system. This means any attack that could pose a possible threat to the information confidentiality, integrity or availability will be considered an intrusion [1].

In anomaly-based intrusion detection systems (AIDSs), a normal model of the behavior of a computer

system is created using machine learning, statistical-based or knowledge-based methods. Any significant deviation between the observed behavior and the model is regarded as an anomaly, which can be interpreted as an intrusion [1].

1.2.2 The UNSW-NB15 dataset

The UNSW-NB15 dataset is based upon virtual traffic generated by IXIA using three different virtual servers. Two servers generate normal traffic, while one server generates attacks. Every record consists of flow features, content features, time features, basic features, general purpose features, connection features [2]. These features originate from different sources at different moments in time but concern the network traffic generated when e.g. making a specific request to a server.

Furthermore each record has a label indicating its intrusion type, when the record is no intrusion the label is Normal. Intrusion labels include Fuzzers, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms. At this moment it is not yet important what these types of attacks are about.

The dataset consists of four csv-files, furthermore there is a file containing the feature names. The UNSW-NB15 dataset has multiple types of features: binary, categorical, numerical.

- Categorical data includes state, service and proto features. The proto feature indicates the transaction protocol, the state feature indicates the state (ACC, CLO, CON, ...) and its dependent protocol and the service feature indicates the service (http, ftp, ...) used.
- label and attack_cat features are target variables.
- srcip is the source ip addresss, dstip is the destination ip address, sport is the source port, dsport is the destination port. These features shouldn't really have any influence on the label or attack_cat and won't be part of a ML model.
- is_sm_ips_ports, is_ftp_login are binary features. When ftp-login is used than is_ftp_login is True, else False. is_sm_ips_ports is True when source and destination ip addresses and port numbers are equal

Numerical features will be addressed in depth in section 2.1, categorical features in section 2.3.

The dataset is located here until 15 december 2023.

1.2.3 Read, inspect and clean the data

Firstly, the code in listing Listing 1.1 is responsible for reading in the data from the unsw-nb15 csv-files and concatenating them in one pandas dataframe, it also adds column names. Column names are read from a separate file on lines 2-5, some processing of the column names is done on line 3-5, the resulting names are added when reading the files. Reading a files happens on line 8, furthermore converters and dtype settings are used to mitigate some problems with unexpected/missing values in cells. The whole reading happens in a loop (started on line 7), and eventually all files are concatenated on line 10. The last line is responsible for removing duplicates. unsw_data() returns a list of file names on line 7.

Listing 1.1: Reading the data

```
raw_data_list = []
column_names = pd.read_csv(external_data_dir() + "/" + 'UNSW-NB15_features.csv',
    ↪ encoding='ISO-8859-1')
column_names['Name'] = column_names['Name'].str.strip()
column_names['Name'] = column_names['Name'].str.lower()
column_names['Name'] = column_names['Name'].apply(lambda x: correct_col_names()[x]
    ↪ if x in correct_col_names() else x)
```

```python

for filename in unsw_data():
 raw_data_part = pd.read_csv(external_data_dir() + "/" + filename, sep=",", header=
    ↪ None, names=column_names['Name'], dtype={'sport': 'string', 'dsport': 'string'
    ↪ , 'attack_cat': 'string'}, converters={'ct_ftp_cmd': empty_string_to_nan, '
    ↪ is_ftp_login': empty_string_to_nan, 'ct_flw_http_mthd': empty_string_to_nan},
    ↪ encoding='ISO-8859-1')
 raw_data_list.append(raw_data_part)
raw_data = pd.concat(raw_data_list)
raw_data.drop_duplicates(inplace=True, subset=None, keep='first')
```

Data is never clean or complete, especially complex data, attack names don't match, NaN values should be replaced,.... So the code in listing Listing 1.2 is applied.

Listing 1.2: Clean data

```python
raw_data['attack_cat'] = raw_data['attack_cat'].replace(np.nan, 'Normal')
raw_data['attack_cat'] = raw_data['attack_cat'].str.strip()
raw_data['attack_cat'] = raw_data['attack_cat'].replace('Backdoor', 'Backdoors')
```

To inspect the data the using info on your dataset like raw_data.info() will give you some info on your dataset.

1.2.4 First impression of the data

To get an idea of the number of attacks for each category present in the dataset see Figure 1.1.

attack_cat	count
Normal	1959771
Exploits	27599
Generic	25378
Fuzzers	21795
Reconnaissance	13357
DoS	5665
Analysis	2184
Backdoors	1983
Shellcode	1511
Worms	171

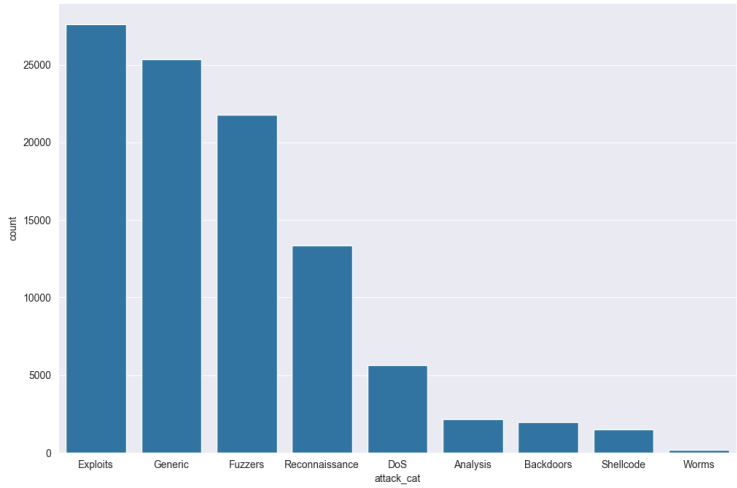

Figure 1.1: Attack counts

Chapter 2

Inspection and preprocessing

2.1 Numerical features

To get an idea of the features included in the UNSW-NB15 dataset, here are some key values (kurtosis,skewness, ...) for every individual numerical feature and a seaborn.kdeplot showing the distribution. A kernel density estimate plot is a method for visualizing the distribution of observations in a dataset, analogous to a histogram. The dataset is very imbalanced, by using a kdeplot, at least there will be something visual if numbers are very low. The normaltest, included in scipy, is used for doing the normaltesting. The test and training set, prepared by the author of [2] don't contain the features 'srcip', 'sport', 'dstip', 'dsport' and it seems like a good idea to exclude them from my dataset from the start to. These features represent the network flow source and destination ip addresses and port numbers. For this dataset all the attacks come from one source server, this is very different from reality. Blacklisting malicious server ip addresses should be done at the firewall level. The features are divided using the same categories as used in [2]. [2] contains more information on the methods used to generate the dataset, the data generated,..... In the description field sometimes numbers will be used to refer to other features, these numbers are used in the UNSW-NB15_features.csv file.

2.1.1 Basic features

dur

name	dur
description	Record total duration
max_value	8786.637695
min_value	0.000000
kurtosis	175.784051
skewness	12.150055
normal	False
nunique_values	587303

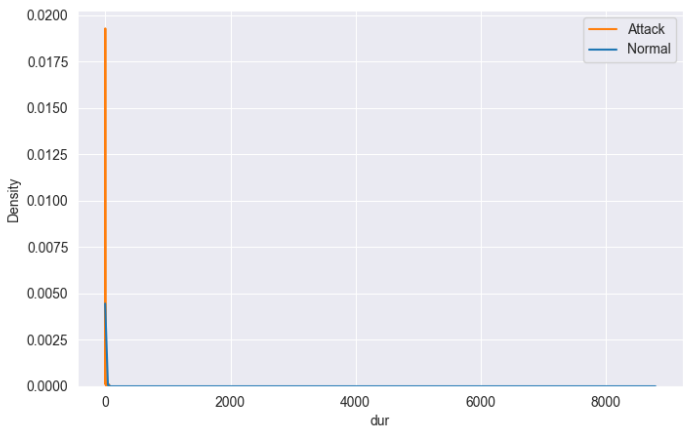

Figure 2.1: Kdeplot dur

sbytes

name	sbytes
description	Source to destination transaction bytes
max_value	14355774
min_value	0
kurtosis	33827.248343
skewness	170.194968
normal	False
nunique_values	14155

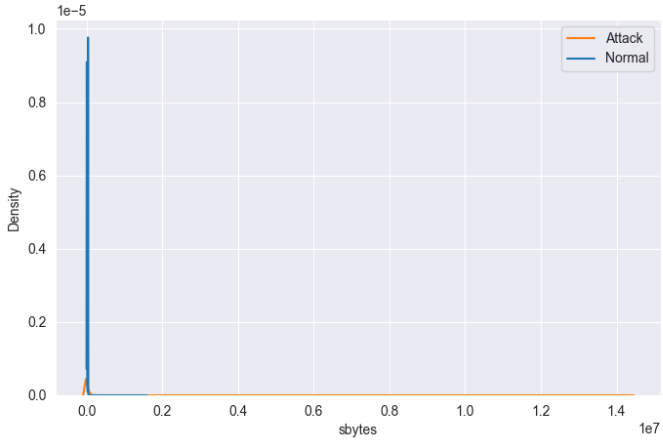

Figure 2.2: Kdeplot sbytes

dbytes

name	dbytes
description	Destination to source transaction bytes
max_value	14657531
min_value	0
kurtosis	302.894366
skewness	9.377388
normal	False
nunique_values	19166

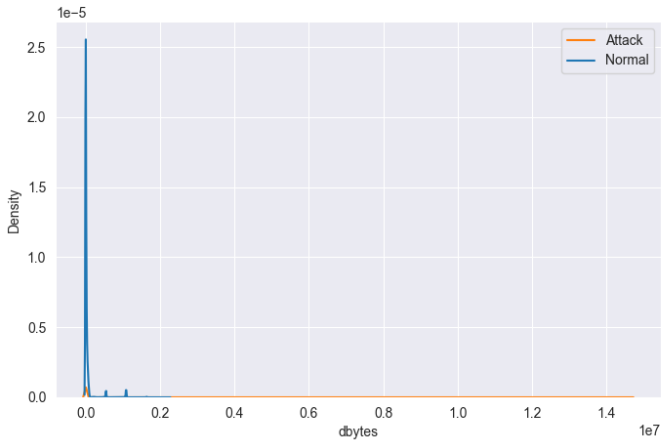

Figure 2.3: Kdeplot dbytes

sttl

name	sttl
description	Source to destination time to live value
max_value	255
min_value	0
kurtosis	13.372484
skewness	3.899301
normal	False
nunique_values	13

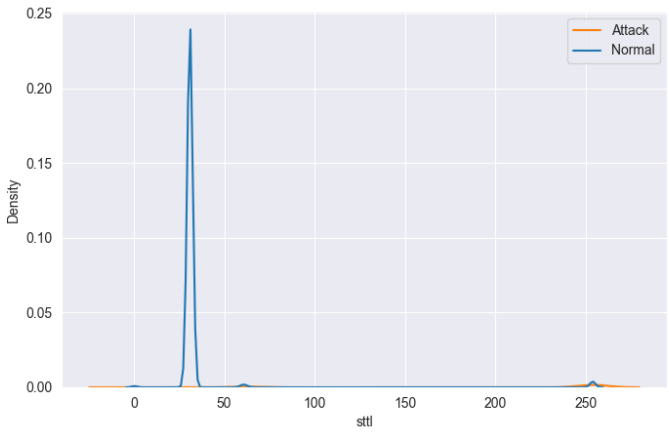

Figure 2.4: Kdeplot sttl

dttl

name	dttl
description	Destination to source time to live value
max_value	254
min_value	0
kurtosis	19.833891
skewness	4.610046
normal	False
nunique_values	11

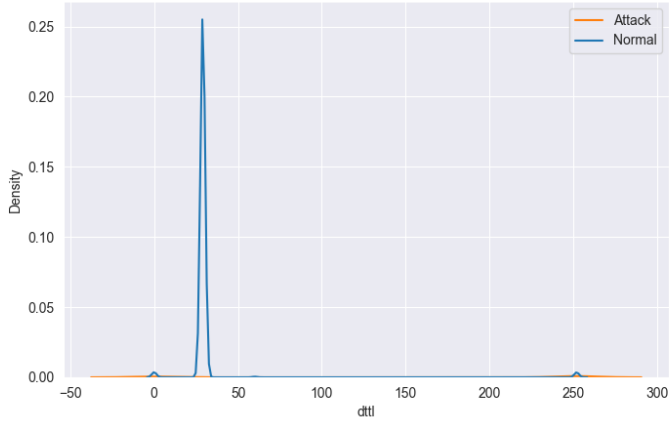

Figure 2.5: Kdeplot dttl

sloss

name	sloss
description	Source packets retransmitted or dropped
max_value	5319
min_value	0
kurtosis	24498.473439
skewness	134.418845
normal	False
nunique_values	544

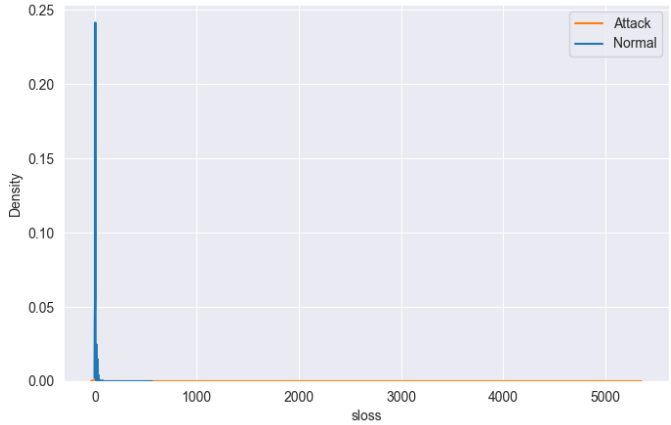

Figure 2.6: Kdeplot sloss

dloss

name	dloss
description	Destination packets retransmitted or dropped
max_value	5507
min_value	0
kurtosis	381.167435
skewness	9.928306
normal	False
nunique_values	707

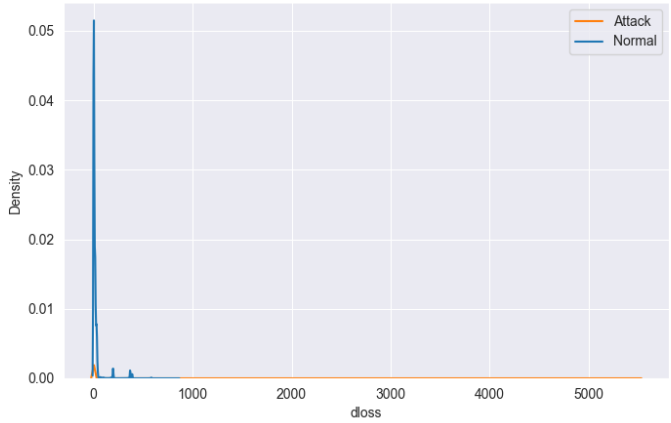

Figure 2.7: Kdeplot dloss

sload

name	sload
description	Source bits per second
max_value	5988000256.000000
min_value	0.000000
kurtosis	719.820680
skewness	20.650483
normal	False
nunique_values	1100258

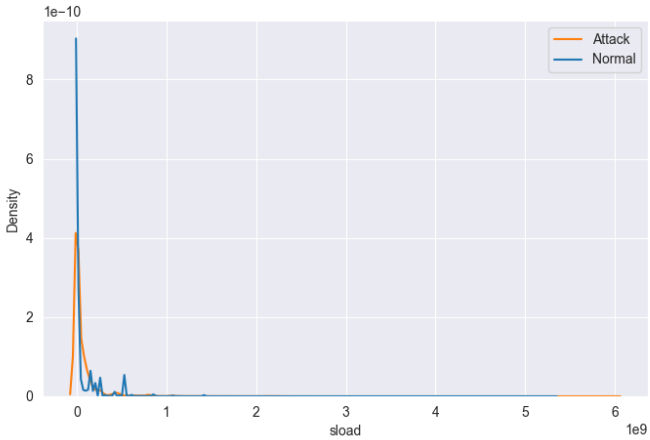

Figure 2.8: Kdeplot Sload

dload

name	dload
description	Destination bits per second
max_value	128761904.000000
min_value	0.000000
kurtosis	1.505539
skewness	1.661193
normal	False
nunique_values	1173739

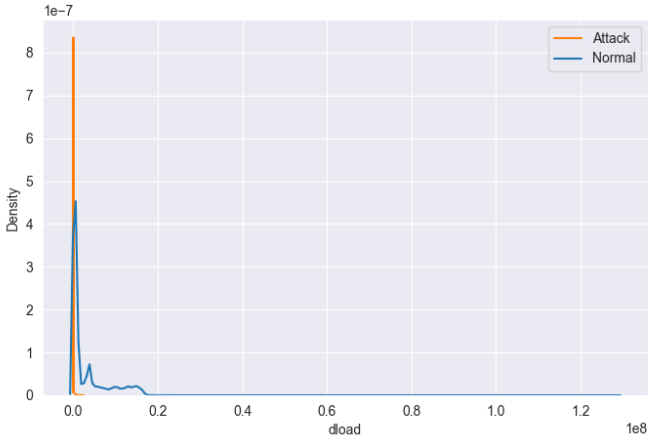

Figure 2.9: Kdeplot Dload

spkts

name	spkts
description	Source to destination packet count
max_value	10646
min_value	0
kurtosis	2210.041851
skewness	24.904787
normal	False
nunique_values	933

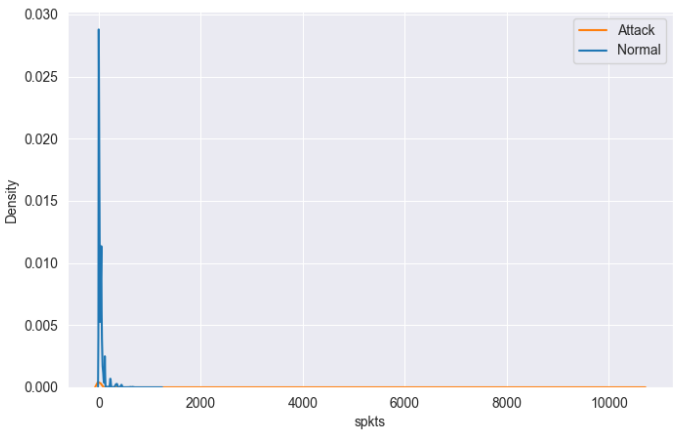

Figure 2.10: Kdeplot Spkts

dpkts

name	dpkts
description	Destination to source packet count
max_value	11018
min_value	0
kurtosis	299.504629
skewness	8.909800
normal	False
nunique_values	1255

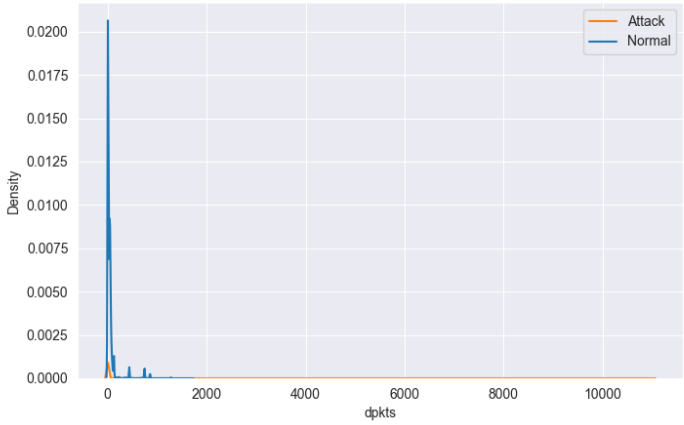

Figure 2.11: Kdeplot Dpkts

2.1.2 Content features

swin

name	swin
description	Source TCP window advertisement value
max_value	255
min_value	0
kurtosis	-1.214775
skewness	-0.886113
normal	False
nunique_values	36

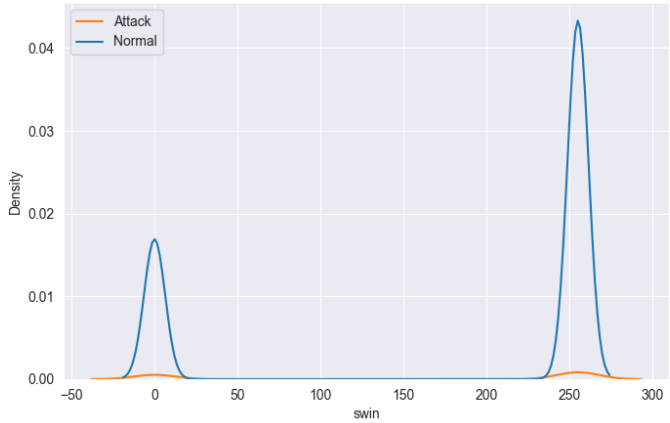

Figure 2.12: Kdeplot swin

dwin

name	dwin
description	Destination TCP window advertisement value
max_value	255
min_value	0
kurtosis	-1.228552
skewness	-0.878311
normal	False
nunique_values	32

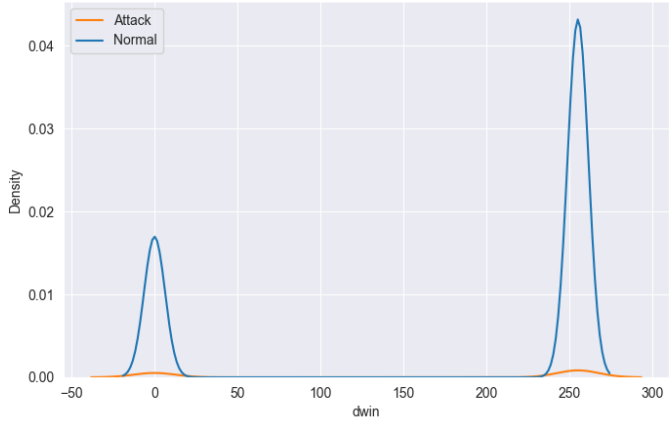

Figure 2.13: Kdeplot dwin

stcpb

name	stcpb
description	Source TCP base sequence number
max_value	4294958913
min_value	0
kurtosis	-1.258301
skewness	0.420820
normal	False
nunique_values	1429571

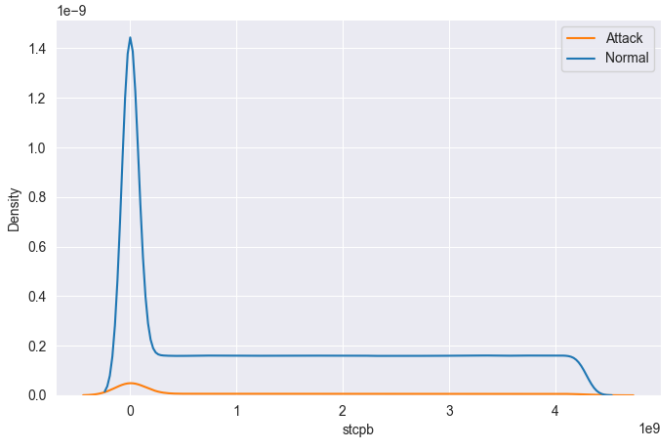

Figure 2.14: Kdeplot stcpb

dtcpb

name	dtcpb
description	Destination TCP base sequence number
max_value	4294953724
min_value	0
kurtosis	-1.249611
skewness	0.426593
normal	False
nunique_values	1429031

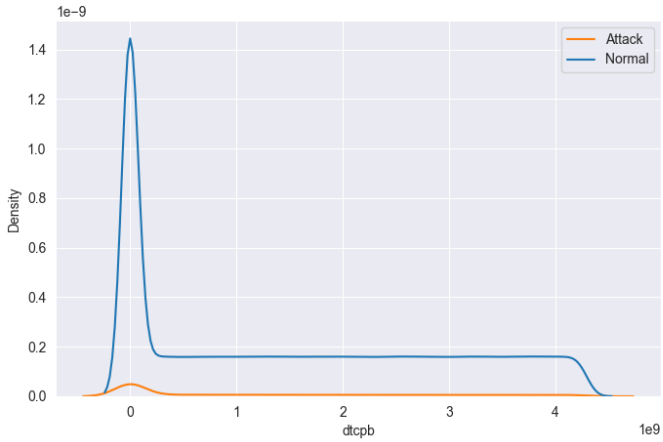

Figure 2.15: Kdeplot dtcpb

smean

name	smean
description	Mean of the flow packet size transmitted by the src
max_value	1504
min_value	0
kurtosis	11.107180
skewness	3.358512
normal	False
nunique_values	1415

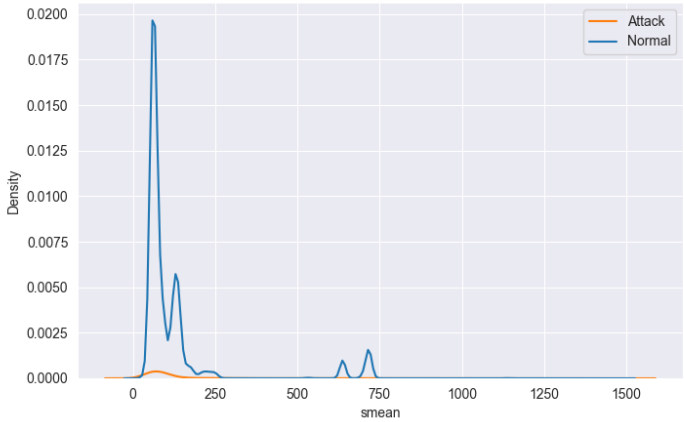

Figure 2.16: Kdeplot smean

dmean

name	dmean
description	Mean of the flow packet size transmitted by the dst
max_value	1500
min_value	0
kurtosis	0.652833
skewness	1.179467
normal	False
nunique_values	1417

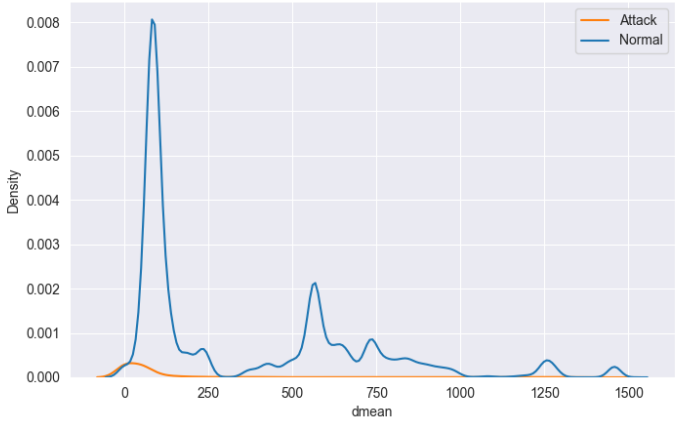

Figure 2.17: Kdeplot dmean

trans_depth

name	trans_depth
description	Represents the pipelined depth into the connection of http request/response transaction
max_value	172
min_value	0
kurtosis	9.162746
skewness	2.963216
normal	False
nunique_values	14

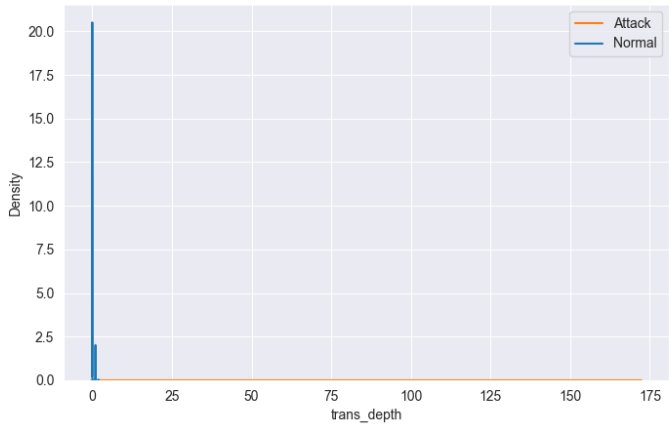

Figure 2.18: Kdeplot trans_depth

res_body_len

name	res_body_len
description	Actual uncompressed content size of the data transferred from the server's http service.
max_value	6558056
min_value	0
kurtosis	145.364860
skewness	11.065744
normal	False
nunique_values	2907

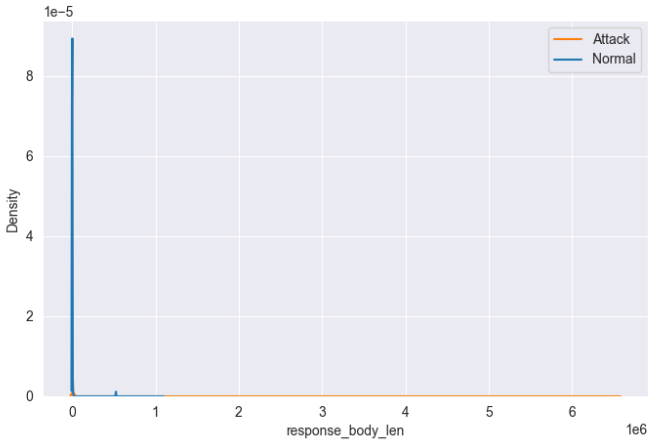

Figure 2.19: Kdeplot res_body_len

2.1.3 Time features

sjit

name	sjit
description	Source jitter (mSec)
max_value	1483830.917000
min_value	0.000000
kurtosis	4251.828235
skewness	58.651216
normal	False
nunique_values	1381001

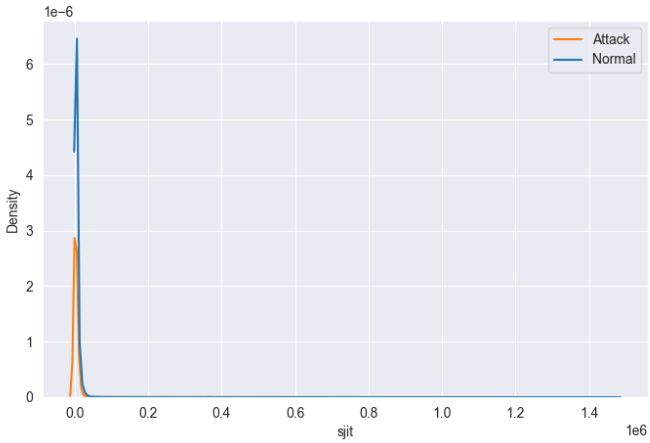

Figure 2.20: Kdeplot djit

djit

name	djit
description	Destination jitter (mSec)
max_value	781221.118300
min_value	0.000000
kurtosis	16155.249096
skewness	84.276305
normal	False
nunique_values	1424689

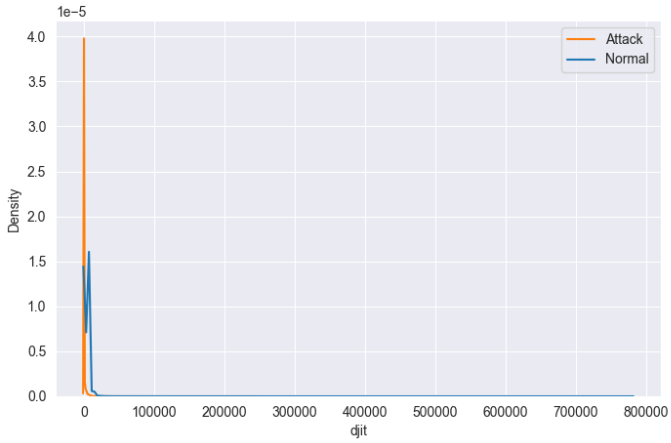

Figure 2.21: Kdeplot djit

stime

name	stime
description	record start time
max_value	1424262068
min_value	1421927377
kurtosis	-1.999332
skewness	-0.013590
normal	False
nunique_values	85348

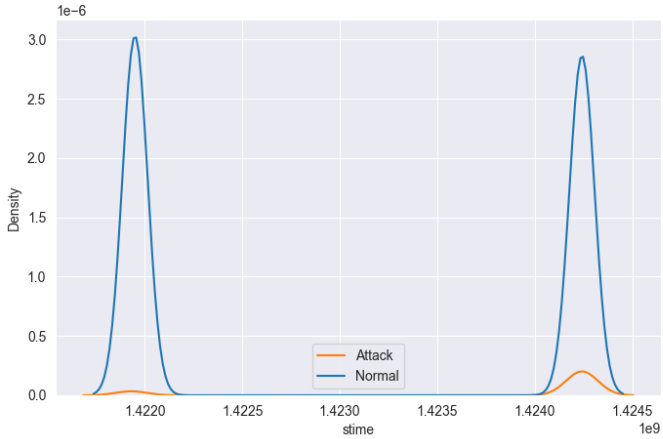

Figure 2.22: Kdeplot stime

ltime

name	ltime
description	record last time
max_value	1424262069
min_value	1421927414
kurtosis	-1.999332
skewness	-0.013590
normal	False
nunique_values	85361

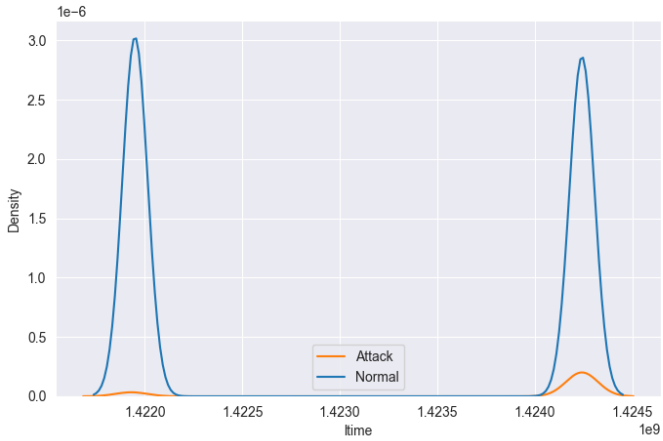

Figure 2.23: Kdeplot ltime

sinpkt

name	sinpkt
description	Source interpacket arrival time (mSec)
max_value	84371.496000
min_value	0.000000
kurtosis	450.574737
skewness	20.631782
normal	False
nunique_values	975275

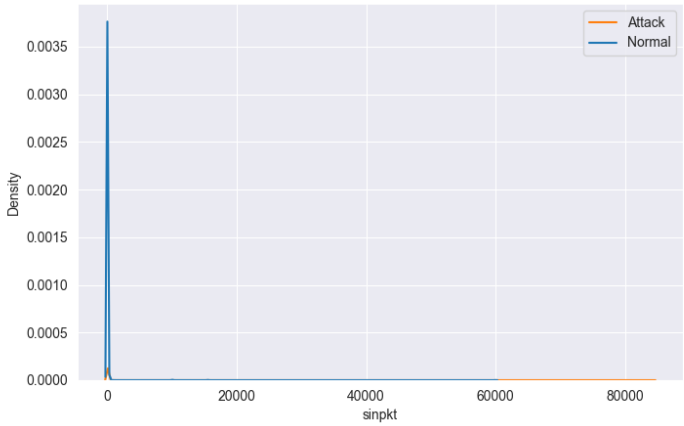

Figure 2.24: Kdeplot sinpkt

dinpkt

name	dinpkt
description	Destination interpacket arrival time (mSec)
max_value	59485.320000
min_value	0.000000
kurtosis	918.645704
skewness	29.082392
normal	False
nunique_values	960353

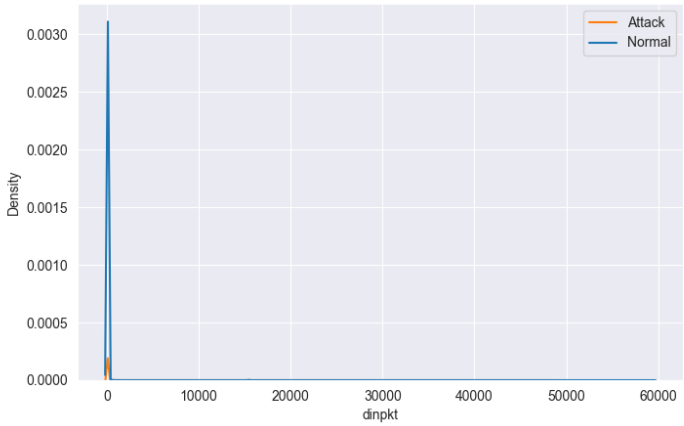

Figure 2.25: Kdeplot dinpkt

tcprtt

name	tcprtt
description	TCP connection setup round-trip time, the sum of synack and ackdat.
max_value	10.037506
min_value	0.000000
kurtosis	1736.210562
skewness	30.438294
normal	False
nunique_values	76931

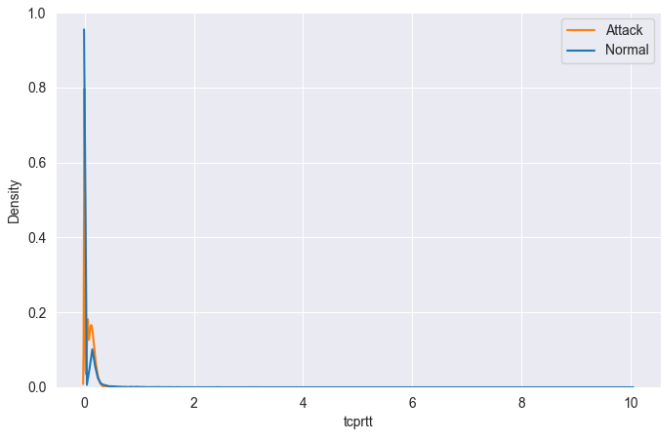

Figure 2.26: Kdeplot tcprtt

synack

name	synack
description	TCP connection setup time, the time between the SYN and the SYN_ACK packets.
max_value	4.525272
min_value	0.000000
kurtosis	3012.466155
skewness	41.387226
normal	False
nunique_values	66997

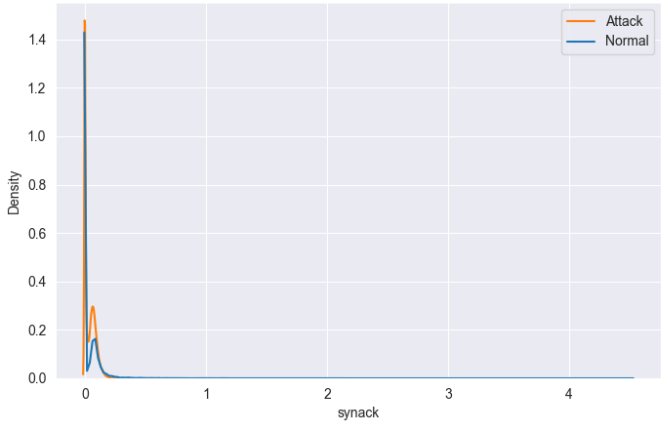

Figure 2.27: Kdeplot synack

ackdat

name	ackdat
description	TCP connection setup time, the time between the SYN_ACK and the ACK packets.
max_value	5.512234
min_value	0.000000
kurtosis	2328.182208
skewness	35.267856
normal	False
nunique_values	60844

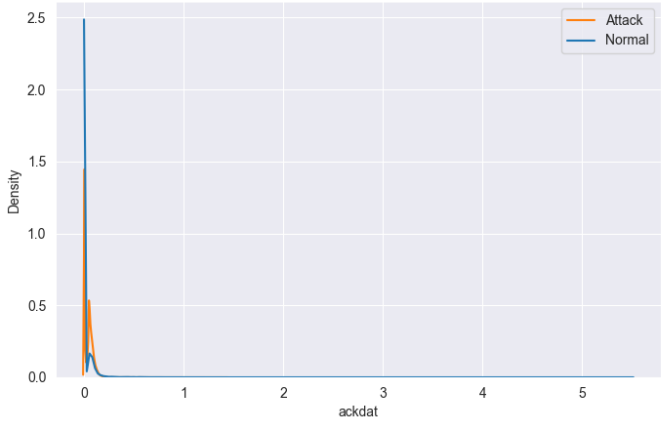

Figure 2.28: Kdeplot ackdata

2.1.4 Additional generated features

ct_state_ttl

name	ct_state_ttl
description	No. for each state (6) according to specific range of values for source/destination time to live (10) (11).
max_value	6
min_value	0
kurtosis	52.318113
skewness	6.071776
normal	False
nunique_values	7

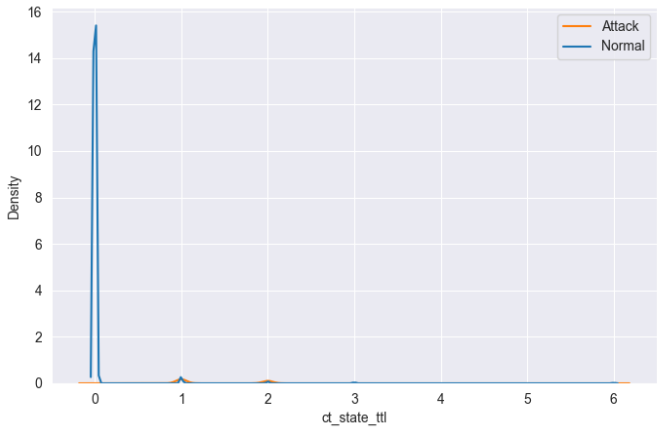

Figure 2.29: Kdeplot ct_state_ttl

ct_flw_http_mthd

name	ct_flw_http_mthd
description	No. of flows that has methods such as Get and Post in http service.
max_value	36
min_value	0
kurtosis	577.125934
skewness	14.778606
normal	False
nunique_values	16

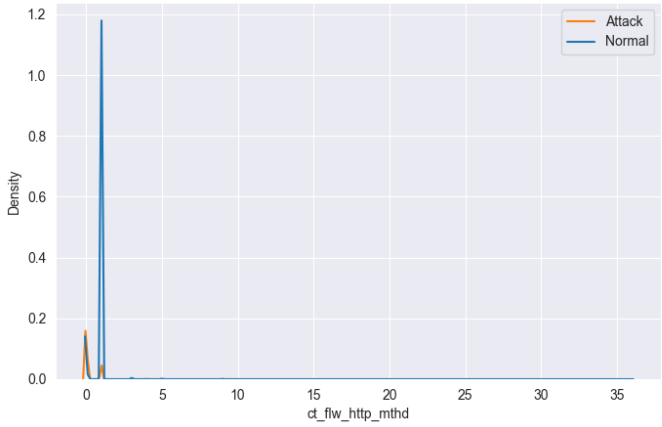

Figure 2.30: Kdeplot ct_flw_http_mthd

ct_ftp_cmd

name	ct_ftp_cmd
description	No of flows that has a command in ftp session.
max_value	8
min_value	0
kurtosis	214.636860
skewness	10.940543
normal	False
nunique_values	8

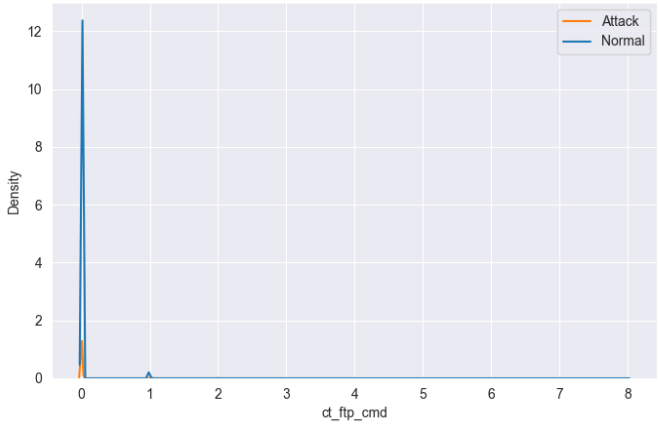

Figure 2.31: Kdeplot ct_ftp_cmd

2.1.5 Connection features

ct_srv_src

name	ct_srv_src
description	No. of connections that contain the same service (14) and source address (1) in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	16.022108
skewness	3.080532
normal	False
nunique_values	65

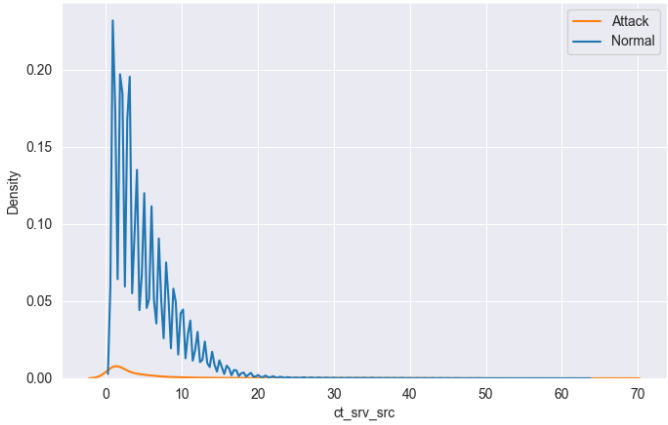

Figure 2.32: Kdeplot ct_srv_src

ct_srv_dst

name	ct_srv_dst
description	No. of connections that contain the same service (14) and destination address (3) in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	18.767795
skewness	3.351385
normal	False
nunique_values	66

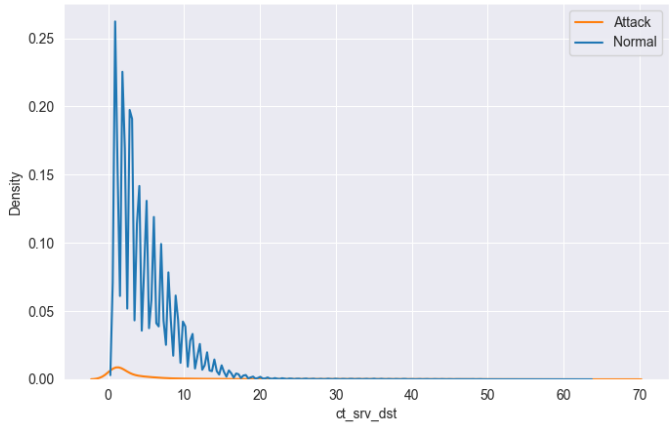

Figure 2.33: Kdeplot ct_srv_dst

ct_dst_ltm

name	ct_dst_ltm
description	No. of connections of the same destination address (3) in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	40.639103
skewness	4.769001
normal	False
nunique_values	62

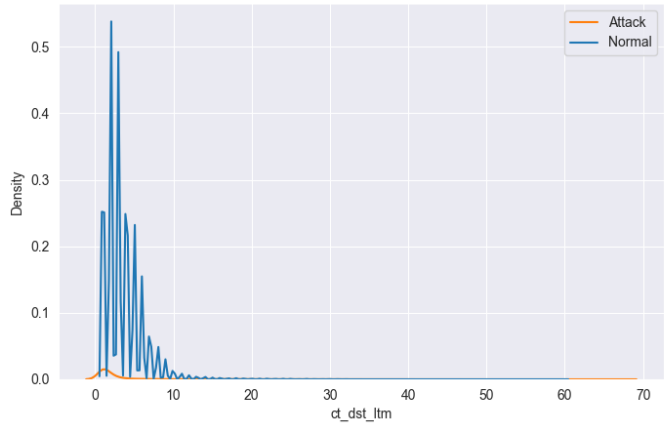

Figure 2.34: Kdeplot ct_dst_ltm

ct_src_ltm

name	ct_src_ltm
description	No. of connections of the same source address (1) in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	28.977552
skewness	4.000931
normal	False
nunique_values	62

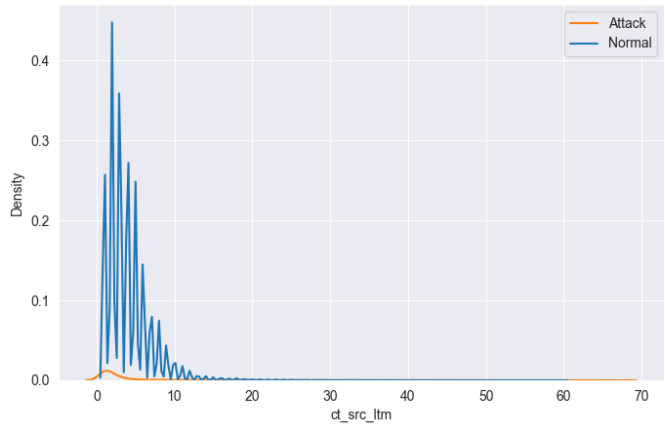

Figure 2.35: Kdeplot ct_src_ltm

ct_src_dport_ltm

name	ct_src_dport_ltm
description	No of connections of the same source address (1) and the destination port (4) in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	107.628780
skewness	9.048341
normal	False
nunique_values	62

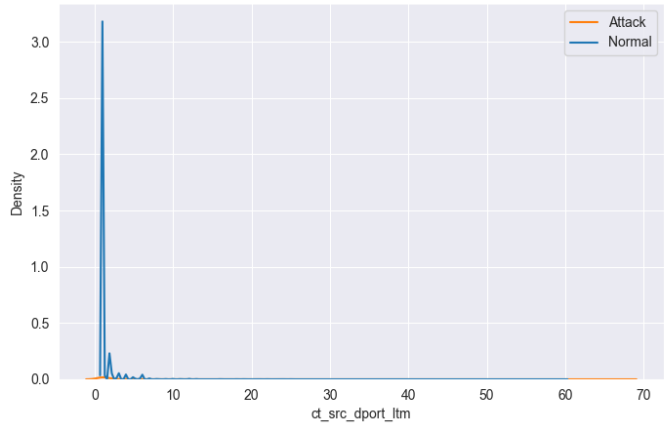

Figure 2.36: Kdeplot ct_src_dport_ltm

ct_dst_sport_ltm

name	ct_dst_sport_ltm
description	No of connections of the same destination address (3) and the source port (2) in 100 connections according to the last time (26).
max_value	60
min_value	1
kurtosis	114.261607
skewness	9.802835
normal	False
nunique_values	55

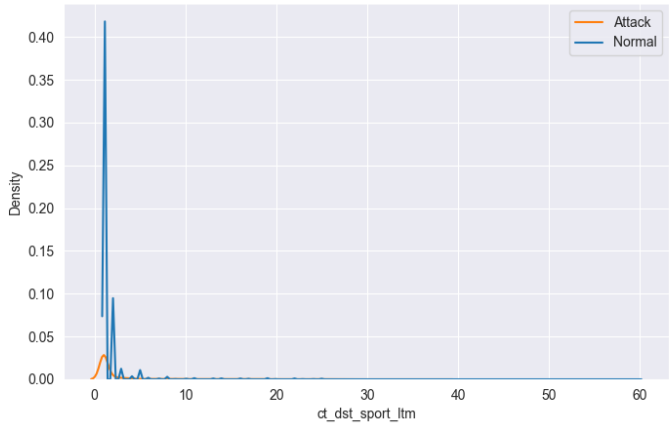

Figure 2.37: Kdeplot ct_dst_sport_ltm

ct_dst_src_ltm

name	ct_dst_src_ltm
description	No of connections of the same source (1) and the destination (3) address in in 100 connections according to the last time (26).
max_value	67
min_value	1
kurtosis	55.741641
skewness	6.741163
normal	False
nunique_values	65

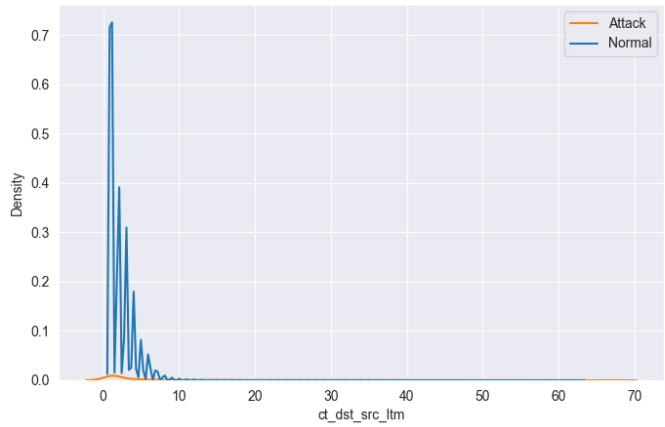

Figure 2.38: Kdeplot ct_dst_src_ltm feature

2.1.6 Code to inspect numerical feature set

To get the key values for every numerical feature and its kdeplot, Python came to the rescue. To inspect the numerical features in the dataset see Listing 2.1, on line 4 a dictionary containing the feature name and the accompanying description gets created, and on line 7 a dataset containing only numerical features is created. Line 8 loops through the columns of the dataset and inspects the column values to create min value and max value, to do a normaltest, to calculate the kurtosis and skewness for the column and so on.

Listing 2.1: Code to inspect numerical feature set

```
1 column_props = pd.DataFrame()
2 columns_info = pd.read_csv(external_data_dir() + "/" + 'UNSW-NB15_features.csv',
    ↪ encoding='ISO-8859-1')
3 columns_info['Name'] = columns_info['Name'].str.strip()
4 name_desc_dict = dict(zip(columns_info.Name, columns_info.Description))
5 c = numeric_features(raw_data)
6 rd = raw_data[c].copy()
7 for feature_name, feature_values in rd.items():
8  column_props = feature_props(feature_values, column_props, feature_name,
    ↪ name_desc_dict[feature_name], 250000)
```

In Listing 2.2 on line 3 a feature properties dictionary gets created and filled. From line 7 up to line 9 a kdeplot gets created, before being saved on line 10. A kernel density estimate (KDE) plot is a method for visualizing the distribution of observations in a dataset. The feature properties are also being converted to latex, this code is omitted.

Listing 2.2: Code to inspect an individual numerical feature

```
1 def feature_props(raw_data, column_values, column_props_global, column_name,
    ↪ column_description, size):
2  column_values_sampled = column_values.sample(size, random_state=0)
3  feat_props = {'name': column_name, 'description': column_description, 'max_value':
    ↪ column_values.max(), 'min_value': column_values.min(), 'kurtosis': scipy.stats
    ↪ .kurtosis(column_values_sampled, bias=True), 'skewness': scipy.stats.skew(
    ↪ column_values_sampled, bias=True), 'normal': scipy.stats.normaltest(
    ↪ column_values_sampled)[1] > 0.5, 'nunique_values': column_values.nunique()}
4  column_props_global = pd.concat([column_props_global, pd.DataFrame([feat_props])],
    ↪ ignore_index=True)
5  column_props_global.to_excel(read_prepare_dir() + '/' + 'column-props.xlsx')
6
7  plt.figure(figsize=(8, 5))
8  sns.kdeplot(x=raw_data[column_name], hue=raw_data['label'], legend=True)
9  plt.legend(labels=['Attack', 'Normal'])
10 plt.savefig(read_prepare_figs_dir() + '/' + column_name.replace('_', '-') + '-hist.
    ↪ png')
11 return column_props_global
```

2.2 Handling numerical features

A machine learning classification algorithm tries to separate different classes of data, so making sure numeric data for all features are standardized is of utmost importance. At this moment the data has mostly numeric columns, to make sure every column in the dataset has the same effect on the target variable, the numeric features need to be 'standardized'. The code in Listing 2.3 does just that. Line 2 produces a dataframe that contains the standardized numeric columns. Basically it uses the scaler from line 1 to transform the raw_data dataset's numeric features, to eventually produce a standardized pandas dataframe in line 6, containing all numeric and non numeric features. As a remark, pandas dataframes have indexes, to remove them reset_index is called on them, line 4 and 5.

It should be noted the function call numeric_features(raw_data) on line 2, returns a list of columns containing numeric features, where non_numeric_features() on line 6 returns a list of non numeric features.

As a sidenote, the sklearn.MinMaxScaler is part of the sklearn package. It transforms the values for every column in such a way that the maximum becomes 1 and the minimum becomes 0.

Listing 2.3: Standardizing numerical data

```
scaler = MinMaxScaler()
raw_data_numeric_std = pd.DataFrame(data=scaler.fit_transform(raw_data[
    ↪ numeric_features(raw_data)]),
columns=numeric_features(raw_data))
raw_data_numeric_std.reset_index(inplace=True)
raw_data.reset_index(inplace=True)
raw_data = pd.concat([raw_data[non_numeric_features()], raw_data_numeric_std], axis
    ↪ =1)
```

2.3 Handling categorical features

Machine learning is all about numbers, and the columns 'proto', 'service' and 'state' don't represent numbers. One way to make sure the categorical values do represent numbers is to create a new binary feature value for every unique value in these columns 'proto', 'service' and 'state'. That would lead to a dataset containing more than 200 features. That would be unfeasible for a computationally limited setup. A way to reduce features is to group columns that have an 'equal' influence on the result of the classification. Not assuming an imbalance between all categories of data (Normal, Fuzzers, Backdoors, ...) is in my opinion a good starting point to estimate this influence.

2.3.1 Grouping categorical values

Before showing the code to group category values in Listing 2.4, some words and figures to make sure you understand the motivation.

Grouping explanation

To reduce the number of unique categorical values, grouping them is in order. As an IDS is all about deciding whether a row in the dataset is to be classified as some kind of attack or not, interest goes to making a crosstab between the categorical values and the attack category. A normalized table expressing the relation between the 'state' values and the attack category should show the way to go for this grouping, see Figure 2.39. To be complete a balanced dataset, meaning a dataset containing an equal proportion of records for every attack_cat value, is used for making the crosstab table.

state	Analysis	Backdoors	DoS	Exploits	Fuzzers	Generic	Normal	Reconnaissance	Shellcode	Worms
ACC	0	0	0	0,35	0,4	0	0,25	0	0	0
CLO	0	0	0	0	0	0	0,53	0,47	0	0
CON	0,01	0,08	0,1	0,04	0	0,01	0,71	0,01	0	0,03
ECO	0	0	0	0	0	0	1	0	0	0
FIN	0,04	0,03	0,11	0,17	0,13	0,02	0,14	0,1	0,1	0,17
INT	0,18	0,18	0,09	0,02	0,07	0,2	0	0,11	0,12	0,03
MAS	0	0	0	0	0	0	1	0	0	0
PAR	0	0	0	0	0	0	1	0	0	0
REQ	0,18	0,37	0,14	0,03	0,14	0,03	0,05	0,06	0	0
RST	0	0	0,37	0,16	0	0,07	0,24	0,15	0	0
TST	0	0	0	0	0	0	1	0	0	0
TXD	0	0	0	0	0	0	1	0	0	0
URH	0	0	0	0	0	0	1	0	0	0
URN	0	0	0	0	0	0	1	0	0	0
no	0	0	0	0	0	0	1	0	0	0

Figure 2.39: Probability distribution by attack category for every unique state value

From Figure 2.39 it is clear that every row containing an integer 1 for the column value Normal should be grouped. Also other rows that have an (almost) identical distribution are eligible for grouping. To make rows that are almost identical identical, a multiplication and usage of the ceil function is necessary. The result from these operations is shown in Figure 2.40, it is clear that the rows that need grouping have equal probability distribution values now. Grouping them should be fairly easy now.

state	Analysis	Backdoors	DoS	Exploits	Fuzzers	Generic	Normal	Reconnaissance	Shellcode	Worms
ACC	0	0	0	4	4	0	3	0	0	0
CLO	0	0	0	0	0	0	6	5	0	0
CON	1	1	1	1	0	1	8	1	0	1
ECO	0	0	0	0	0	0	10	0	0	0
FIN	1	1	2	2	2	1	2	1	1	2
INT	2	2	1	1	1	2	0	2	2	1
MAS	0	0	0	0	0	0	10	0	0	0
PAR	0	0	0	0	0	0	10	0	0	0
REQ	2	4	2	1	2	1	1	1	0	0
RST	0	0	4	2	0	1	3	2	0	0
TST	0	0	0	0	0	0	10	0	0	0
TXD	0	0	0	0	0	0	10	0	0	0
URH	0	0	0	0	0	0	10	0	0	0
URN	0	0	0	0	0	0	10	0	0	0
no	0	0	0	0	0	0	10	0	0	0

Figure 2.40: A probability distribution by attack category for every unique state value

The result of grouping should look like Figure 2.41. This figure shows one group is formed.

Analysis	Backdoors	DoS	Exploits	Fuzzers	Generic	Normal	Reconnaissance	Shellcode	Worms	state
0	0	0	0	0	0	6	5	0	0	['CLO']
0	0	0	0	0	0	10	0	0	0	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']
0	0	0	4	4	0	3	0	0	0	['ACC']
0	0	4	2	0	1	3	2	0	0	['RST']
1	1	1	1	0	1	8	1	0	1	['CON']
1	1	2	2	2	1	2	1	1	2	['FIN']
2	2	1	1	1	2	0	2	2	1	['INT']
2	4	2	1	2	1	1	1	0	0	['REQ']

Figure 2.41: Result of the grouping statement

Grouping code

Line 7 in Listing 2.4 is responsible for effectively grouping the state categories. This line is similar to a "group by" in sql statements, it aggregates the rows, and uses a lambda statement to make a list of the column values that will form a group.

Listing 2.4: Reduce categorical columns

```
1  for feature in features_to_be_denominalized():
2   ct = pd.crosstab(raw_data[feature], raw_data['attack_cat'], normalize='index').
      ↪ round(2)
3   ct = ct.multiply(10)
4   ct = ct.apply(np.ceil)
5   l = ct.columns.to_list()
6   ct = ct.reset_index()
7   result = ct.groupby(l, as_index=False).agg({feature: lambda x: list(x)})
8   df = pd.DataFrame()
9   df = pd.DataFrame({'feat': result[feature], 'feat_agg': result[feature]})
10  df = df.explode('feat')
```

```
11 df['column_name'] = df.apply(lambda x: create_column_name(x), axis=1)
12 feat_dict = dict(zip(df.feat, df.column_name))
```

To replace the value of columns in a pandas dataframe the easiest thing to do is to create a dictionary, lines 8-12 of Listing 2.4 are responsible for creating this dictionary. This dictionary should consist of the column name to be replaced and the new column name.

On Listing 2.4 line 11 creates the new column name using the function "create_column_name" defined in Listing 2.5. It does so by applying a lambda function to every row in dataframe df. The new column name will stay the same as the original name, if no grouping was appropriate. If only two values were grouped then these two values are used for the new column name, otherwise use two values trailed by an ellipsis.

The dataframe df adapted on line 11 looks like Figure 2.42, the first and third column will eventually be used to create a dictionary in line 12 Listing 2.4.

feat	feat_agg	column_name
CLO	['CLO']	CLO
ECO	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
MAS	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
PAR	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
TST	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
TXD	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
URH	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
URN	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
no	['ECO', 'MAS', 'PAR', 'TST', 'TXD', 'URH', 'URN', 'no']	ECO--MAS...
ACC	['ACC']	ACC
RST	['RST']	RST
CON	['CON']	CON
FIN	['FIN']	FIN
INT	['INT']	INT
REQ	['REQ']	REQ

Figure 2.42: Almost a dictionary

Listing 2.5: Create column name

```
def create_column_name(row):
 if len(row['feat_agg']) == 1:
  return row['feat_agg'][0]
 elif len(row['feat_agg']) == 2:
  return row['feat_agg'][0] + '--' + row['feat_agg'][1]
 elif len(row['feat_agg']) > 2:
  return row['feat_agg'][0] + '--' + row['feat_agg'][1] + '...'
```

Renaming categorical features in the global dataframe The previously created dictionaries will be used to rename the column values by using Listing 2.6. The function call features_be_denominalized() returns a list ["proto", "service", "state"].

Listing 2.6: Replacing values of categorical columns for columns "state", "service" and "proto"

```
for feature in features_to_be_denominalized():
 raw_data = raw_data.replace({feature: feat_dict})
```

Code to get a balanced dataset The code to create a balanced dataset to base the probability distributions on is shown in Listing 2.7. This code loops through every type of data (Normal, Backdoors, Fuzzers, ...) in the dataset and takes some data, if not enough unique records are available for a attack category, records will be resampled. Resetting the index and dropping it on line 11 makes every record unique, even

though the dataset generated potentially contains duplicates as we are resampling with **replace** set to True. Especially when testing the model against a classifier using KFold cross validation while not resetting the index prevented a correct stratification, especially where attacks were low. The balanced dataset as created here will always be the same when using the same dataset. This is especially important when retesting the model after it changed, and also when comparing classifiers.

Listing 2.7: Code to create a balanced dataset

```
raw_data_tmp = raw_data.copy()
X_complete = pd.DataFrame()
for attack_cat in raw_data_tmp.attack_cat.unique():
 tmp_attack_cat = raw_data_tmp[raw_data_tmp['attack_cat'] == attack_cat]
 if len(tmp_attack_cat) >0:
  if len(tmp_attack_cat) < size / len(raw_data_tmp.attack_cat.unique()):
   X_attack_cat = resample(tmp_attack_cat, replace=True, n_samples=int(size / len(
    ↪ raw_data_tmp.attack_cat.unique())), random_state=0)
  else:
   X_attack_cat = resample(tmp_attack_cat, replace=False, n_samples=int(size / len(
    ↪ raw_data_tmp.attack_cat.unique())), random_state=0)
 X_complete = pd.concat([X_complete, X_attack_cat])
X_complete = X_complete.reset_index(level=None, drop=True, inplace=False, col_level
    ↪ =0, col_fill='')
return X_complete
```

Some thoughts By grouping column values that share more or less the same probability distribution, considering their association with respective attack categories, the total feature set for the time being is reduced from above 200 to less than 70 for the dataset at hand. By doing this, the probability distributions are somehow 'fixed' within the dataset. If for some reason these distributions should change, one should be aware of this, for this might have an impact on classification results.

Grouped proto values The service feature showed no possibilities for grouping, but the proto feature did, as can be seen in Figure 2.43

proto
['arp', 'icmp', 'igmp', 'rtp']
['udp']
['tcp']
['sctp']
['ospf']
['ipv6', 'sep']
['mobile', 'pim', 'sun-nd', 'swipe']
['3pc', 'a/n', 'aes-sp3-d', 'any', 'argus', 'aris', 'ax.25', 'bbn-rcc', 'bna', 'br-sat-mon', 'cbt', 'cftp', 'chaos', 'compaq-peer', 'cphb', 'cpnx', 'crtp', 'crudp', 'dcn', 'ddp', 'ddx', 'dgp', 'egp', 'eigrp', 'emcon', 'encap', 'etherip', 'fc', 'fire', 'ggp', 'gmtp', 'gre', 'hmp', 'i-nlsp', 'iatp', 'ib', 'idpr', 'idpr-cmtp', 'idrp', 'ifmp', 'igp', 'il', 'ip', 'ipcomp', 'ipcv', 'ipip', 'iplt', 'ipnip', 'ippc', 'ipv6-frag', 'ipv6-no', 'ipv6-opts', 'ipv6-route', 'ipx-n-ip', 'irtp', 'isis', 'iso-ip', 'iso-tp4', 'kryptolan', 'l2tp', 'larp', 'leaf-1', 'leaf-2', 'merit-inp', 'mfe-nsp', 'mhrp', 'micp', 'mtp', 'mux', 'narp', 'netblt', 'nsfnet-igp', 'nvp', 'pgm', 'pipe', 'pnni', 'pri-enc', 'prm', 'ptp', 'pup', 'pvp', 'qnx', 'rdp', 'rsvp', 'rvd', 'sat-expak', 'sat-mon', 'sccopmce', 'scps', 'sdrp', 'secure-vmtp', 'skip', 'sm', 'smp', 'snp', 'sprite-rpc', 'sps', 'srp', 'st2', 'stp', 'tcf', 'tlsp', 'tp++', 'trunk-1', 'trunk-2', 'ttp', 'unas', 'uti', 'vines', 'visa', 'vmtp', 'vrrp', 'wb-expak', 'wb-mon', 'wsn', 'xnet', 'xns-idp', 'xtp', 'zero']

Figure 2.43: Grouped proto values

2.3.2 Categorical values as binary features

As the dataset has had its categorical columns reduced, it's a good time to create the 'dummy' columns for the remaining categorical values. The pandas package is equipped with a function that returns a dataframe containing the extra columns.

Listing 2.8: Categorical data as binary columns

```
raw_data = pd.get_dummies(raw_data, columns=["proto"])
raw_data = pd.get_dummies(raw_data, columns=["service"])
raw_data = pd.get_dummies(raw_data, columns=["state"])
```

2.4 Handling features with low variance

Because data that shows almost no variance is practically useless, as it is almost always the same it can't be used as a predictor of some sort, a sklearn.VarianceThreshold is used to select these columns from the dataset. Listing 2.9 is used to identify and remove low variance columns. As there are approximately only 60000 attacks and more than two million records the threshold is set very low. If the threshold is set to zero, all records in the dataset must have the same value for a specific feature before this feature is flagged to be removed. Dur, trans_depth, proto_udt, proto_esp and state_ECR are eligible for deletion under a threshold of 0.00001. After deleting these features from the dataset, there are 69 features left.

On lines 1-4 of Listing 2.9 label and attack_cat columns are 'saved' and deleted from the dataset, on lines 11 and 12 the label and attack_cat columns are reincluded in the dataset.

Listing 2.9: Removing low variance columns

```
1  attack_cat = raw_data.attack_cat.copy()
2  Label = raw_data.label.copy()
3  raw_data = raw_data.drop('attack_cat', axis=1)
4  raw_data = raw_data.drop('label', axis=1)
5
6  var_thr = VarianceThreshold(threshold=0.00001)
7  var_thr.fit(raw_data)
8  to_be_deleted = [x for x in map(lambda x, y: x if not y else None, raw_data.columns,
       ↪  var_thr.get_support()) if x is not None]
9
10 for col in to_be_deleted:
11  raw_data = raw_data.drop(col, axis=1)
12
13 raw_data['attack_cat'] = attack_cat
14 raw_data['label'] = Label
```

2.5 Features included in dataset

The reduced features for our dataset will include the features in Figure 2.44, these features include the grouped dummy categorical features. and exclude some low variance dataset features.

columns	number of columns
is_ftp_login, is_sm_ips_ports,, sbytes, dbytes, sttl, dttl, sloss, dloss, sload, dload, spkts, dpkts, swin, dwin, stcpb, dtcpb, smean, dmean, response_body_len, sjit, djit, stime, ltime, sinpkt, dinpkt, tcprtt, synack, ackdat, ct_state_ttl, ct_flw_http_mthd, ct_ftp_cmd, ct_srv_src, ct_srv_dst, ct_dst_ltm, ct_src_ltm, ct_src_dport_ltm, ct_dst_sport_ltm, ct_dst_src_ltm, proto_3pc—a/n..., proto_arp—icmp..., proto_ipv6—sep, proto_mobile—pim..., proto_ospf, proto_sctp, proto_tcp, proto_udp, service_-, service_dhcp, service_dns, service_ftp, service_ftp-data, service_http, service_irc, service_pop3, service_radius, service_smtp, service_snmp, service_ssh, service_ssl, state_ACC, state_CLO, state_CON, state_ECO—MAS..., state_FIN,state_INT, state_REQ, state_RST, attack_cat, label	69

Figure 2.44: Features for the intermediate dataset

Chapter 3

Zero measuring, adapting and retesting the brand new model

Zero measurement In this chapter the model created in the previous chapter will be tested, let's call this a zero measurement. The performance of the model will be measured. At the moment the dataset has 69 features and 10 target classes, features including 'dummy' binary category features, binary features, numerical features, and the label and the attack_cat features. The classifier used to test our model, will be the decision tree classifier (sklearn.DecisionTreeClassifier). This classifier is used because it doesn't take that long for a gridsearch (sklearn.GridSearchCV) and a performance test using stratified k-fold cross validation (sklearn.StratifiedKFold), for this kind of classifier, to finish. Testing results will precede the code that produced the results.

- **Hyper parameters** for the classifier will be optimized using our model.
- The **balanced multiclass dataset** used for gridsearching, testing and training will count 2000000 instances. It will be balanced by oversampling the attack classes, not only instances of attack classes are mutually imbalanced, there is also an imbalance between attack instances and normal instances.
- **Oversampling** is not ideal, as repeated instances will partly devaluate testing.
- To test for the robustness of the model a **stratified k-fold cross validation** technique will be used. Basically, this technique will split the already balanced dataset into a number of stratified splits. For every split it guarantees an equal distribution of every class, that is what stratification is all about. Each fold is then used once as a validation while the k-1 remaining folds form the training set.

Feature reducing After conducting the zero measurement, the model's features will be reduced by checking for correlation, and using a regularization technique Lasso.

Retesting The reduced model will be tested using the procedure as proposed for the zero measurement.

3.1 Zero measurement

3.1.1 Results

To get an idea of the performance of the model when using the sklearn.DecisionTreeClassifier. Every split of our dataset is tested, and this results in a confusion matrix for every split. To get an overall idea of performance see this consolidated confusion matrix Figure 3.6, composed of every individual confusion matrix. For every class (horizontal row) the matrix shows the classes that it is easily confused with. When the true class of a sample is Analysis, the class with which it is being confused most is Backdoors.

Because attack instances are all being oversampled, this figure is not really honest. Every test instance

is almost always used for training too, and every instance has multiple identical copies against which it is validated. The result of this falseness is to be seen for rows Worms and Shellcode. These classes are nearly perfecly detected. The Normal class is confused most with Fuzzers, when the true label for a record is Normal, on 20000 normal records, it is seen as being a Fuzzer. All in all accuracy is a fairly good for this Normal class.

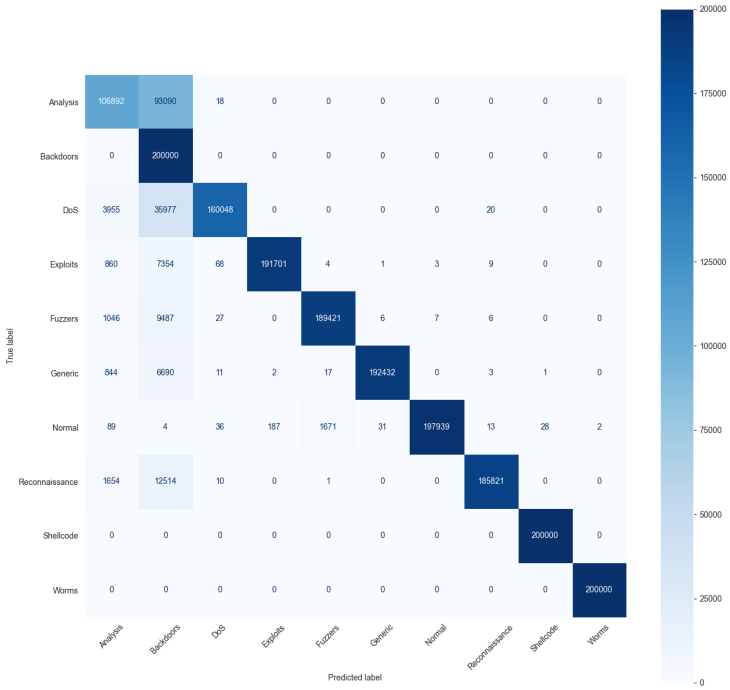

Figure 3.1: Consolidated confusion matrix when cross validating using StratifiedKFold with 10 folds

3.1.2 Code

Start test

The code to get the balanced dataset is already shown elsewhere, see Listing 2.7. The code to test the balanced dataset is in Listing 3.1, at first the classifier and a param_grid is fetched, on lines 3 and 4. These will be used to find the optimal parameters for this classifier using sklearn.GridSearchCV, line 5. Then the cross validation is used to test and evaluate the model using the optimized classifier, lines 7 and 8. To be complete, actually parameter optimization using GridSearchCV or any other hyperparameter tuning algorithm should be done with data that is not used anywhere else, but as the dataset is unbalanced and for some attacks very limited, the data will be reused.

Listing 3.1: Test the dataset

```
1 balanced_data = get_balanced_dataset(raw_data, size)
2 X, y = remove_target_columns(balanced_data, multi=True)
3 clf = get_classifier('dt')
4 param_grid = get_params('dt')
5 opt_params = grid_search(clf, param_grid, X, y)
6 clf_opt = get_classifier('dt', opt_params[0])
7 cv = StratifiedKFold(n_splits=10)
8 evaluate_model_cm(clf_opt, X, y, cv)
```

Doing a gridsearch

A gridsearch for optimal hyperparameters is done by feeding the classifier, the dictionary with possible values for all hyperparameters, setting a cv to the number of folds and setting a scoring parameter into the sklearn.GridSearchCV function, see line 2 of Listing 3.2, and then fitting the gridsearch function to the dataset and its target labels, line 3. This should result in the best parameters being returned, see line 4.

Listing 3.2: Gridsearch

```
1 def grid_search(classifier, param_grid, X, y):
2  optimal_params = GridSearchCV(classifier, param_grid, cv=5, scoring='f1_micro',
     ↪ verbose=0, n_jobs=-1)
3  optimal_params.fit(X, y)
4  opt = pd.DataFrame(optimal_params.best_params_.items())
5  opt.columns = ['param', 'value']
6 return optimal_params.best_params_, opt
```

Getting an accumulated confusion matrix

In Listing 3.3 every fold of the cross validation step results in a new set of train and test data, which in turn is used to train and test the classifier, line 7 and 8. The true values for y and the predicted labels get fed into the confusion_matrix function. The results of this function are added to the conf_mat_accum on line 10. Lines 12-19 will produce the final result as seen previously. Of course other kinds of scoring could be implemented here too.

Listing 3.3: Get an accumulated confusion matrix

```
1 def evaluate_model_cm(clf, X, y, cv):
2  score = np.zeros(cv.get_n_splits([X, y]))
3  conf_mat_accum = np.zeros((y.nunique(), y.nunique()))
4  for train_ix, test_ix in cv.split(X, y):
5   train_x, test_x = X.loc[X.index[train_ix]], X.loc[X.index[test_ix]]
6   train_y, test_y = y.loc[y.index[train_ix]], y.loc[y.index[test_ix]]
7   classifier = clf.fit(train_x, train_y)
8   predicted_labels = classifier.predict(test_x)
9   conf_mat = confusion_matrix(test_y, predicted_labels)
10  conf_mat_accum = np.add(conf_mat_accum, conf_mat)
11
12 plt.grid(False)
13 plt.figure(figsize=(15, 15))
14 ax = plt.subplot(1, 1, 1)
15 ax.grid(False)
16 disp = ConfusionMatrixDisplay(confusion_matrix=conf_mat_accum.astype(int),
     ↪ display_labels=clf.classes_)
17 disp.plot(values_format='', ax=ax, cmap='Blues')
18 plt.xticks(rotation=45)
19 plt.savefig(test_classifiers_figs_dir() + '/' + str(y.nunique()) + '-class-
     ↪ confusion-map.png')
```

3.2 Highly correlated features

To reduce the number of highly correlated features, a table of highly correlated features is created and used to handpick the features that offer almost no extra information. As a result of this high correlation the features stime, ltime, dloss, swin and dpkts are being dropped from the dataset. A heatmap for all numerical and binary features can be found, see Figure A.1.

column	other_column	correlation
sbytes	sloss	0,959443829099997
dbytes	dloss	0,991619788062274
dbytes	Dpkts	0,971750296015398
dloss	Dpkts	0,992238670635247
swin	dwin	0,996474205887993
swin	proto_tcp	0,99995686990766
swin	proto_udp	-0,973791743441161
swin	state_FIN	0,982460649874752
dwin	proto_tcp	0,996459681542453
dwin	proto_udp	-0,970386063398602
dwin	state_FIN	0,985311130105128
Stime	Ltime	0,999999999865909
proto_tcp	proto_udp	-0,973833744980492
proto_tcp	state_FIN	0,982414683264794
proto_udp	state_FIN	-0,956708570127634

Figure 3.2: Highly correlated features

To produce the correlation between columns see Listing 3.4. On line 10 the highly correlated columns along with their correlation are being exported to an excel file.

Listing 3.4: Highly correlated features

```
1 features_corr = pd.DataFrame({'column': [], 'other_column': [], 'correlation': []})
2 corr_mat = raw_data.corr(method='pearson')
3 columns = corr_mat.columns
4 for i in range(corr_mat.shape[0]):
5  for j in range(i + 1, corr_mat.shape[0]):
6   if corr_mat.iloc[i, j] >= 0.95 or corr_mat.iloc[i, j] <= -0.95:
7    new_row = {'column': columns[i], 'other_column': columns[j], 'correlation':
     ↪ corr_mat.iloc[i, j]}
8    features_corr = pd.concat([features_corr, pd.DataFrame([new_row])], ignore_index=
     ↪ True)
9 features_corr.to_excel(external_data_dir() + '/' + 'features_corr.xlsx',
10 index=False)
```

3.3 Retesting

The removal of features results in a confusion matrix as in Figure 3.3.

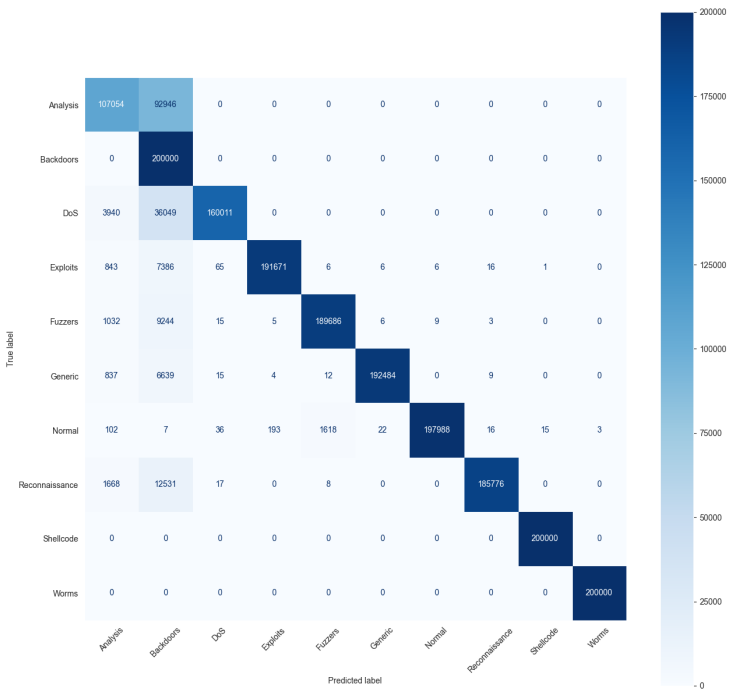

Figure 3.3: Consolidated confusion matrix when cross validating using StratifiedKFold with 10 folds, after reducing correlated features

3.3.1 Performance change after correlation reduction

Most of the changes due to reducing highly correlated features are improvements as can be seen from Figure 3.4.

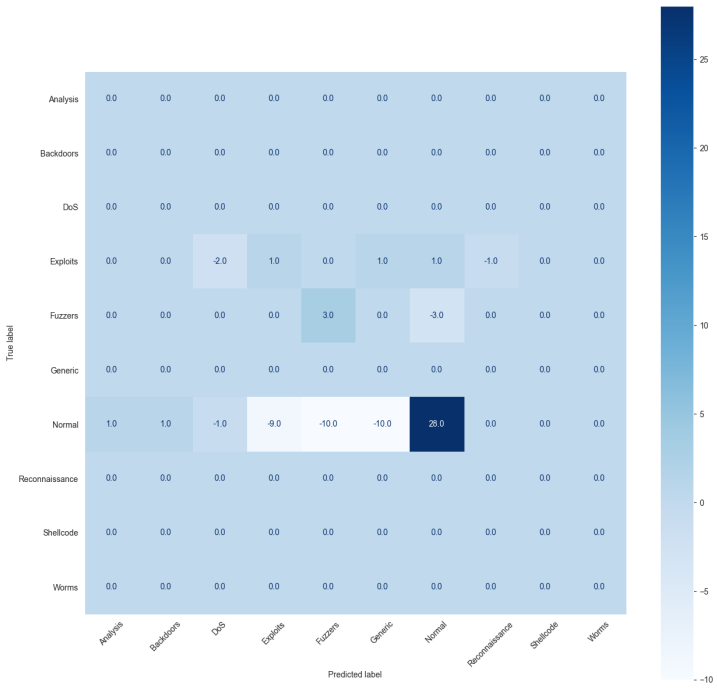

Figure 3.4: Subrtraction of zero measurement confusion matrix and post correlation reduction confusion matrix

3.4 Further reducing features using Lasso

A technique to further reduce features is Lasso, it is based on linear regression. To make sklearn.Lasso work optimally it is important to remove any feature that has a high correlation with another, which already happened. Further more no other non linear relation should be present between pairs of features. Non linear relations are relations like cubic, quadratic, exponential. Lasso is an iterative technique, so it is more optimal to only have features left that are as independent as possible. Lasso aims to minimize the sum of squared residuals between the predicted and actual values, while simultaneously adding a penalty on the magnitude of the regression coefficients.

3.4.1 Pairplots

To check the type of relationship between every pair of numerical features, a scatterplot is a fairly obvious method. The code to produce these plots is quite simple using seaborn, see listing Listing 3.5. Normally pairplots would be produced as a matrix of all figures, this would lead to a matrix as wide as 30 graphs, very inconvenient to show here. So the figures are created per 4 or 5, or less than 5.

Listing 3.5: Produce pairplots using seaborn

```
b_raw_data = get_balanced_dataset(raw_data,  size)
b_raw_data.drop(['Label'], inplace=True, axis=1)
b_raw_data.drop(['attack_cat'], inplace=True, axis=1)
cols = b_raw_data.columns.to_list()
for i in range(ceil(len(b_raw_data.columns) / 4)):
 if len(cols) == 5:
  cols_to_be_shown = cols[:5]
 elif len(cols) >= 4:
  cols_to_be_shown = cols[:4]
```

```
elif len(cols) < 4 and len(cols) > 0:
 cols_to_be_shown = cols
elif len(cols) == 0:
 return

cols = [item for item in cols if item not in cols_to_be_shown]
sns.pairplot(b_raw_data[cols_to_be_shown])
plt.savefig(feature_reduction_dir() + '/figs/columns_pairplot_' + str(i) + '.png')
```

To get an impression of how pairplots look, see Figure 3.5, the diagonal of the matrix shows the histogram for the feature in question. The pairplots show no linear, quadratic, cubic, exponential relationship. The other pairplots are more or less the same, and it has no added value to show them here.

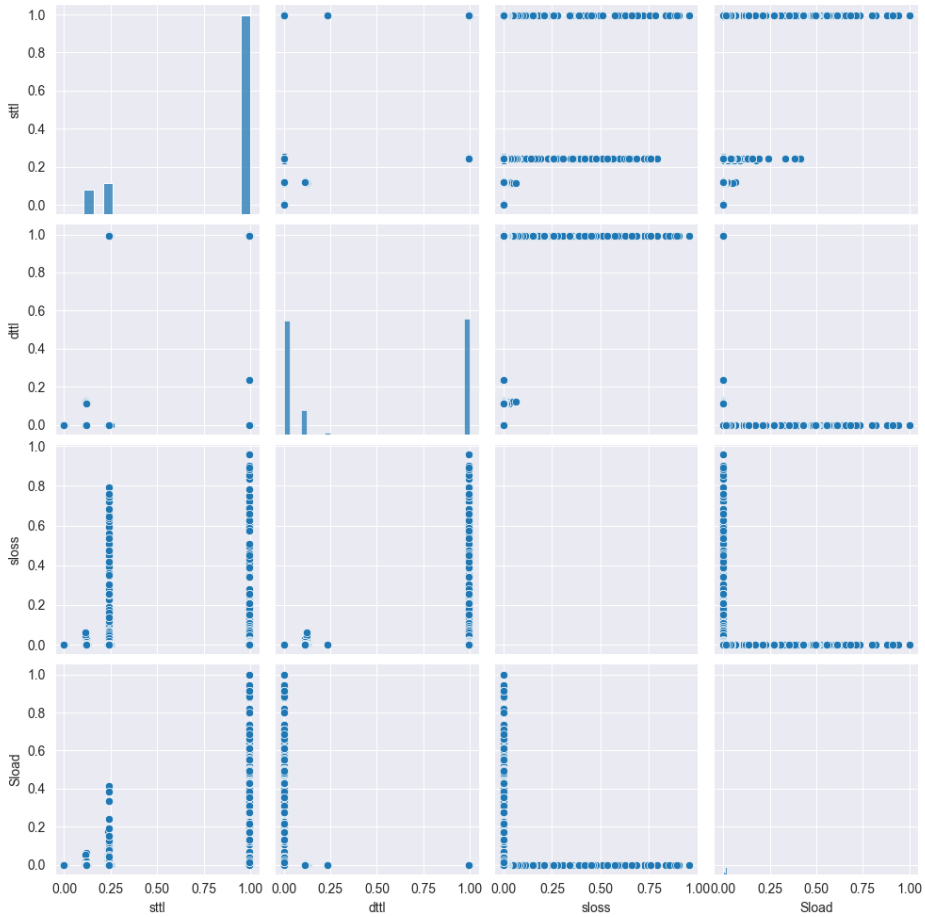

Figure 3.5: Pairplot for first columns

3.4.2 Reduce features using Lasso

To reduce features using Lasso, see Listing 3.6. A multiclass balanced dataset is used, to fit a dataset to a linear model Lasso. A higher value for the alpha parameter will result in reducing more features.

Listing 3.6: Fit data to Lasso

```
balanced_data = get_balanced_dataset(raw_data, 250000)
X, y = remove_target_columns(balanced_data, True)
y = pd.factorize(y)[0].astype(np.uint16)
lasso = Lasso(alpha=0.00001, max_iter=25000, tol=1e-3)
lasso.fit(X, y)
```

Fitting the data to Lasso results in a coefficient for every feature. Those features that are less interesting for our classification will have a coefficient of zero or nearly zero. Other features will have higher coefficients. To get an idea of features that have higher coefficient values see Figure A.2.

Listing 3.7: Showing features to (possibly) be removed

```
def condition(x):
 return x > 0.0000001 or x < -0.0000001

indices_to_be_removed = [idx for idx, element in enumerate(lasso.coef_) if not
    ↪ condition(element)]
features_to_be_removed = X.columns[indices_to_be_removed]
print(features_to_be_removed)
```

Lasso regularization indicates features service_ssl', 'spkts', 'ct_ftp_cmd are candidates to be removed from our dataset. As a test spkts is removed from the dataset.

3.5 Retesting after Lasso reduction

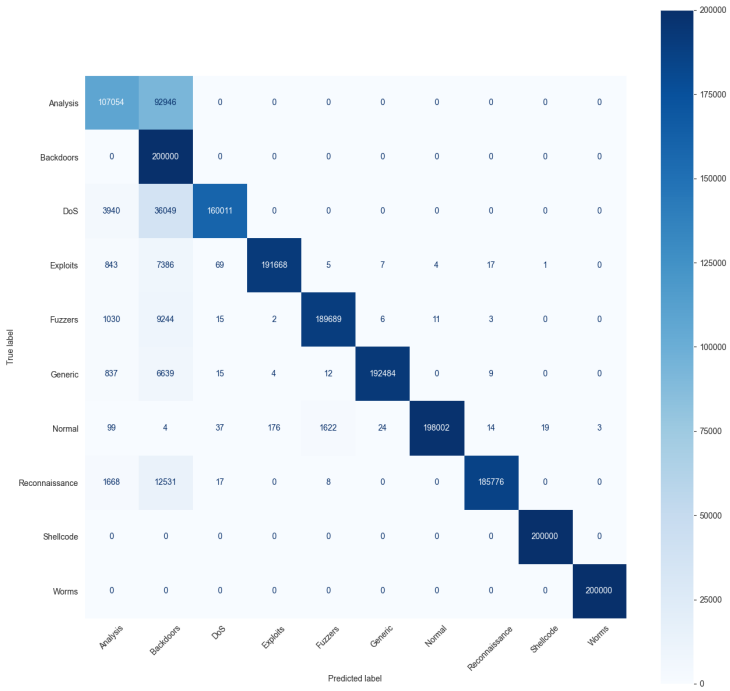

Figure 3.6: Consolidated confusion matrix when cross validating using StratifiedKFold with 10 folds, after correlated feature reduction and LASSO reduction

3.6 Performance change after a partial Lasso reduction

As our classifiers need to consider 9 classes of attacks and 1 normal class and some of the highest correlated features have already been removed. Actually removing the 'spkts' feature is not that interesting as can be seen from Figure 3.7, maybe another feature proposed by the Lasso method returns better results.

Even though the 'spkts' feature shouldn't have been deleted from the dataset, the tests in chapter 4 will be performed on a dataset without this feature.

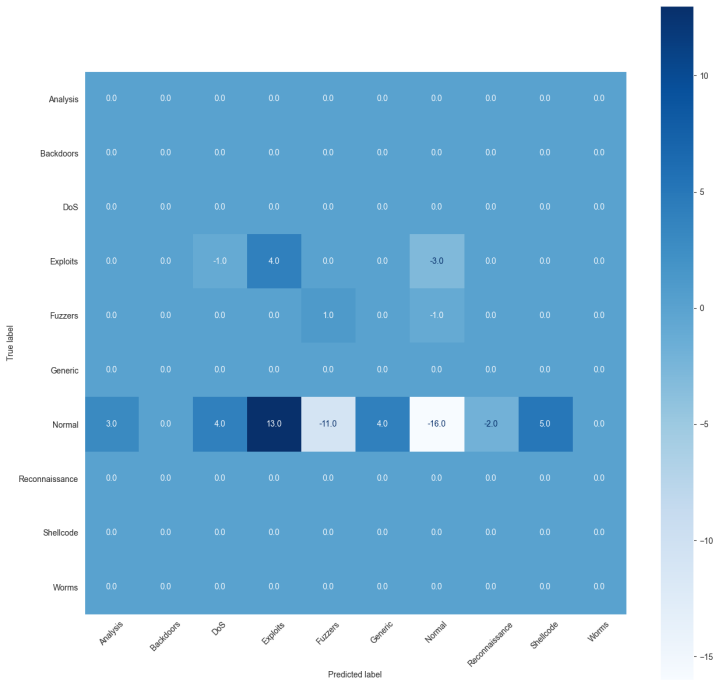

Figure 3.7: Performance change between post correlation reduction confusion matrix and post Lasso reduction confusion matrix

Chapter 4

Comparing the model on different classifiers

To get an idea of the performance of the model a comparison will be made between different classifiers, the support vector classifier (sklearn.SVC), the decision tree classifier (sklearn.DecisionTreeClassifier), k-nearest neighbors classifier (sklearn.KNeighborsClassifier).

4.1 Training and testing several different classifiers

Testing a support vector machine classifier and a logistic regression using 2000000 rows, hyperparameter optimization, doing cross validation is computationally demanding.

So the test here considered a balanced 250000 row dataset. The dataset used was identical for every classifier. Every classifier is trained and tested with the same data for every fold. So differences in accuracy for every fold can only be traced back to differences in qualities of the respective classifier and its hyperparameters.

4.1.1 Results

It is clear from Figure 4.1 that the decision tree classifier is the most accurate at classifying the dataset, by far.

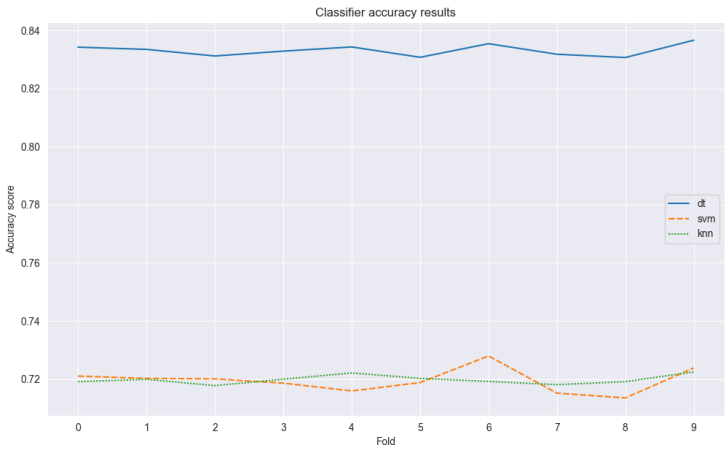

Figure 4.1: Results for a 10-fold cross validation test of a decision tree classifier(dt), a k nearest neighbor(knn) classifier and a support vector machine classifier(svm) using 250000 rows of a balanced dataset with oversampling

The results are bit disappointing, an IDS that has an accuracy of 80% is not to be trusted. It needs to be said that the work done so far is only intended to be a starting point for an IDS. Furthermore, my guess is

that the decision tree classifier benefits the most from oversampling data. Still it is quite remarkable that the classifier that is the fastest when it comes to hyperparameter tuning and training and testing is the most accurate.

4.1.2 Thoughts on hyperparameters

The optimal parameters were entropy for the criterion parameter and 150 for the max_depth parameter when considering the decision tree classifier. The k nearest neighbor classifier has an optimal leaf_size of 1, and a n_neighbors of 3. For the support vector machine based classifier the optimal kernel value is 'rbf', which is no surprise as it is the only possible value. Furthermore its optimal C parameter is 100, and its optimal gamma parameter is 1.

SVM parameters

According to the gridsearch the optimal C parameter value is 100, which is the highest in the grid. May be a even higher value would result in better results. Gamma is high as well. Perhaps the possible parameter values should be adapted in Listing 4.4.

A smaller value of C allows for a larger margin, which in turn could raise the number of misclassifications when training. A large C value emphasizes minimizing the training error, which could result in a narrower margin. A large C value could increase the risk of overfitting.

Gamma is a parameter which needs to be set when the rbf kernel is being chosen. Gamma defines how far the influence of a single training example reaches, with low values for gamma meaning 'far'.

Other hyperparameter tuning algorithms could possibly make the difference.

KNN parameters

As 3 is almost the minimal value for the k_neighbors parameter and optimal leaf_size is 1 which is minimal. I wonder if I should have used some more hyperparameters when gridsearching. Parameters like weight, considering multifeatured datasets can become very sparse, and maybe metric.

DT parameters

As this was the best performing classifier, there is nothing really to say here. However I wonder which features are being used in the decision tree classifier. Just to know which are not interesting for this classifier.

4.1.3 Code

Code to test the model

To be able to test multiple classifiers at a time code in **??** comes in handy. The parameters for this function are the data, a list of classifier abbreviations and the number of rows the tested dataset will contain. When size is set to 250000,

Listing 4.1: Testing the model on different classifiers

```
def test_classifiers_basic(raw_data, kinds, size):
 balanced_data = get_balanced_dataset(raw_data, size)
 optimal_params[kind] = pd.DataFrame()
 X, y = remove_target_columns(balanced_data)

 for kind in kinds:
  clf = get_classifier(kind)
```

```
 param_grid = get_params(kind)
 opt_params = grid_search(clf, param_grid, X, y)
 clf_opt = get_classifier(kind, opt_params[0])
 cv = StratifiedKFold(n_splits=10)
 scores_opt = evaluate_model_scoring(clf_opt, X, y, cv)
```

Code to evaluate the model

The model gets evaluated using cross validation on a number of metrics, as cross validation is used, some of the metrics have to be averaged. Actually all the scoring details ar handled by sklearn, though it is possible to do your own calculations if necessary.

Listing 4.2: Scoring the model on different classifiers

```
def evaluate_model_scoring(classifier, X, y, cv):
 scoring = ['precision_macro', 'recall_macro', 'f1_macro', 'accuracy']
 scores = cross_validate(estimator=classifier, X=X, y=y, cv=cv, scoring=scoring,
   ↪ n_jobs=-1)
 scores = pd.DataFrame(scores)
 return scores
```

Code to get classifiers

The code to get a classifier is to be seen in Listing 4.3. When the hyperparameters for a given classifier are added as a dictionary together with the abbreviation for the classifier, a classifier with the chosen hyperparameters is returned. For example a KNeighborsClassifier (line 10) is returned when 'knn' is used for the 'kind' parameter and no hyperparameters are given. If there is a hyperparameter dictionary passed to the function, the hyperparameters get set on line 12.

Listing 4.3: Get classifier

```
1  def get_classifier(kind, params=None):
2   match kind:
3    case 'svm':
4     if params is None:
5      return svm.SVC()
6     else:
7      return svm.SVC(C=params['C'], gamma=params['gamma'], kernel=params['kernel'])
8    case 'knn':
9     if params is None:
10     return KNeighborsClassifier(n_neighbors=3)
11    else:
12     return KNeighborsClassifier(n_neighbors=params['n_neighbors'], leaf_size=params[
      ↪ 'leaf_size'])
13   case 'dt':
14    if params is None:
15     return DecisionTreeClassifier()
16    else:
17     return DecisionTreeClassifier(criterion=params['criterion'], max_depth=params['
      ↪ max_depth'])
18   case _:
19    return
```

Code to get parameter grid for a classifier

GridSearchCV is a hyperparameter tuning algorithm, it expects a multitude of all possible hyperparameters. When it searches for the best possible combination of hyperparameters for a specific classifier, it does so exhaustively. When doing this kind of optimization for a 10 fold cross validation with 2000000 records, things can take days and weeks especially when it involves many parameter options (on a simple

i5 pc).

For most classifiers some combinations of hyperparameters are not possible. It is best to test for these things using small datasets on beforehand. Some iterative processes need enough data to converge, making things very iterative at times.

Listing 4.4: Get parameter grid for classifier

```
def get_params(kind):
 match kind:
  case 'dt':
   return {'criterion': ['gini', 'entropy'], 'max_depth': [4, 5, 6, 7, 8, 9, 10, 11,
   ↪  12, 15, 20, 30, 40, 50, 70, 90, 120, 150]}
  case 'svm':
   return {'C': [0.5, 1, 10, 100], 'gamma': ['scale', 1, 0.1, 0.01, 0.001, 0.0001],
   ↪ 'kernel': ['rbf']}
  case 'knn':
   leaf_size = list(range(1, 50))
   k_range = list(range(1, 31))
   return dict(n_neighbors=k_range, leaf_size=leaf_size)
  case _:
   return
```

Code for the graph

The results that are produced when evaluating the model are a pandas dataframe, they look like Figure 4.2 when exported to excel.

dataset	clf	score	0	1	2	3	4	5	6	7	8	9
Balanced	dt	fit_time	7.3	7.12	7.12	7.2	7.15	7.21	7.11	7.34	6.72	6.69
Balanced	dt	score_time	0.49	0.49	0.48	0.5	0.48	0.5	0.49	0.49	0.46	0.46
Balanced	dt	test_precision_macro	0.86	0.86	0.86	0.86	0.87	0.86	0.86	0.86	0.86	0.86
Balanced	dt	test_recall_macro	0.83	0.83	0.83	0.83	0.83	0.83	0.84	0.83	0.83	0.84
Balanced	dt	test_f1_macro	0.84	0.84	0.84	0.84	0.84	0.84	0.84	0.84	0.84	0.85
Balanced	dt	test_accuracy	0.83	0.83	0.83	0.83	0.83	0.83	0.84	0.83	0.83	0.84
Balanced	svm	fit_time	4465.8	4400.6	4393.7	4425.11	4333.51	4366.83	4415.64	4309.6	3781.0	3794.09
Balanced	svm	score_time	534.77	530.21	543.33	543.41	530.6	551.42	545.09	529.88	484.86	486.16
Balanced	svm	test_precision_macro	0.74	0.74	0.74	0.74	0.74	0.74	0.75	0.74	0.74	0.74
Balanced	svm	test_recall_macro	0.72	0.72	0.72	0.72	0.72	0.72	0.73	0.72	0.71	0.72
Balanced	svm	test_f1_macro	0.72	0.72	0.72	0.72	0.72	0.72	0.73	0.72	0.71	0.72
Balanced	svm	test_accuracy	0.72	0.72	0.72	0.72	0.72	0.72	0.73	0.72	0.71	0.72
Balanced	knn	fit_time	0.56	0.57	0.57	0.56	0.59	0.6	0.6	0.52	0.5	0.49
Balanced	knn	score_time	39.19	39.03	39.0	38.98	38.89	38.91	38.96	39.06	38.3	38.26
Balanced	knn	test_precision_macro	0.8	0.8	0.8	0.81	0.81	0.81	0.8	0.81	0.8	0.81
Balanced	knn	test_recall_macro	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72
Balanced	knn	test_f1_macro	0.7	0.71	0.7	0.7	0.71	0.71	0.7	0.7	0.7	0.71
Balanced	knn	test_accuracy	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72

Figure 4.2: Results

To show only a graph for the accuracy, the dataset needs to be filtered, see line 1 of Listing 4.5, then the columns 'dataset' and 'score' need to be dropped, line 2 and 3. The data needs to be transposed, line 4. The graph to be produced gets its data set on line 9.

Listing 4.5: Results lineplot

```
1 results = results.query("score == 'test_accuracy'")
2 results = results.droplevel(0, axis=0)
3 results = results.droplevel(1, axis=0)
4 results = results.T
5 plt.figure(figsize=(12, 7))
6 plt.xlabel('Fold')
7 plt.ylabel('Accuracy score')
```

```
plt.xticks(results.index.to_list())
sns.lineplot(data=results).set_title('Classifier accuracy results')
plt.legend(title='')
plt.savefig(figures_dir() + '/' + 'results.png')

```

Conclusion

Actually the UNSW-NB15 dataset was more of an excuse to start this machine learning project than a goal in itself.

This report handles several things, from reading in the data, over inspecting its features, over handling different types of data, over different techniques to reduce features, to ways to score the model and to compare some basic machine learning classifiers. Of course not every action I took made it to this report. Sometimes the search is more interesting than the finding.

Being fairly uninitiated in cybersecurity, I have the feeling the attack set is quite limited compared to the set of normal network traffic. This dataset limitation hinders me in meeting modern machine learning requirements.

Creating the software project and finding a model that fitted the setting are iterative processes by nature. So I had to do and redo several things several times before every function acted as expected. Add to that the fact that I am relatively new to machine learning as a whole, so it is fairly obvious to see that it was quite a task to write this report, even though many things in my report have already been done before. Nevertheless I am proud to have given some insight into the world of machine learning, especially for the machine learning novice.

That is why I added some words on ideas to be explored in the next chapter of this writing.

Future work

Even though I am happy with the software project as conceived for writing this report, it still isn't the universal machine learning dataset exploring and testing tool I would want it to be. Nevertheless here are some leads to follow to make my work on the UNSW-NB15 dataset more complete.

Exploring the pristine dataset using a decision tree algorithm

Doing some initial exploration of the complete dataset using the decision tree classifier could have lead to a better insight in the features and its split values used by it.

Reduce features by combining them in one

Some features could be kept but reduced by subtracting them, like stime and ltime. But there are a lot more. Other kinds of combinations could of course make sense too.

Explain individual examples

Try to explain individual predictions. LIME, the acronym for local interpretable model-agnostic explanations, is a technique that approximates any black box machine learning model with a local, interpretable model to explain each individual prediction.

Pairwise class membership testing

Find features that add most to the separability of two classes.

Handle outliers

Find out which examples in the dataset are outliers and try to find a way to deal with them, or use the information that can be extracted from them.

Improve hyperparameter tuning

Seeing that both SVM and KNN didn't seem to perform as well as DT, maybe part of the problem is in the hyperparameters and its values. I maybe should have used more parameters.

Use an ensemble learning technique like random forest

As the decision tree classifier was already performing quite well, I wonder how a random forest would perform.

Visualize dataset to represent highly similar classes

As the whole idea of intrusion attacks is about mimicking normal traffic, it would be nice to visually see which kind of attacks mostly resemble normal traffic, which attacks have a lot in common with other attacks. Umap is a tool for visualizing multi-featured multi-classed datasets. It would make a good starting point for finding features that make the difference between two or more classes.

Find a better way to handle the imbalanced dataset

Oversampling the minority classes is not the only way of dealing with imbalanced datasets. It is the simplest way, but SMOTE (Synthetic Minority Oversampling Technique) could be a better alternative to plain oversampling.

Nesting classifiers

The performance of the decision tree classifier tested and trained with a balanced dataset of 2000000 rows was very high, considering deciding whether a record was normal or not. Most of the accuracy loss had to do with classifying one attack as another, so nesting classifiers to firstly decide if a record is an attack and then deciding on the attack type, could make it possible to improve accuracy on both levels.

Make a neural network to use as a classifier for our dataset

Make a neural network that acts as a classifier for our dataset.

Bibliography

[1] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1):1–22, 2019.

[2] 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)*, pages 1–6, 2015. doi: 10.1109/MilCIS.2015.7348942.

Appendix A

Extras

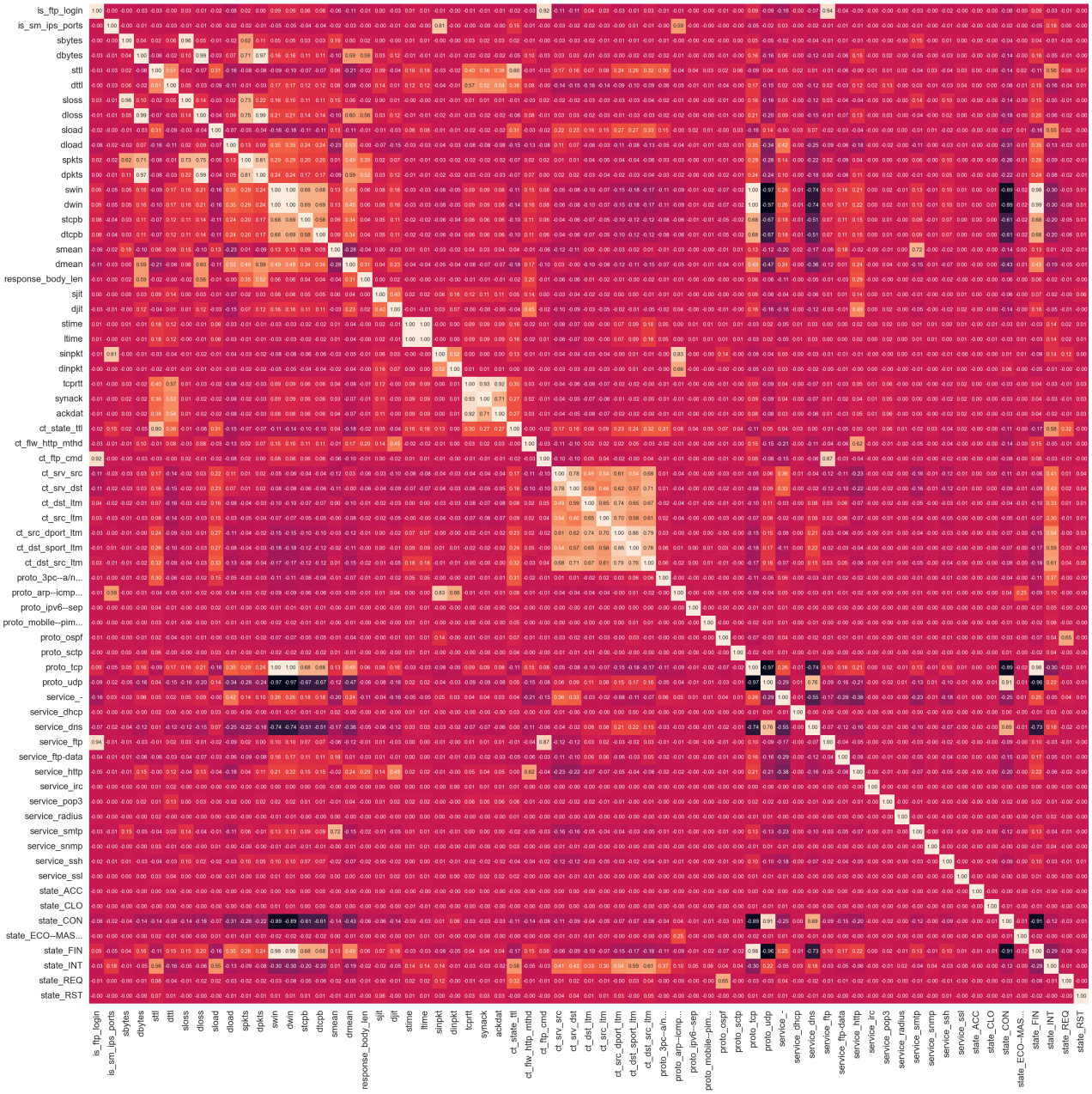

Figure A.1: Heatmap that shows the pairwise correlations between numerical features

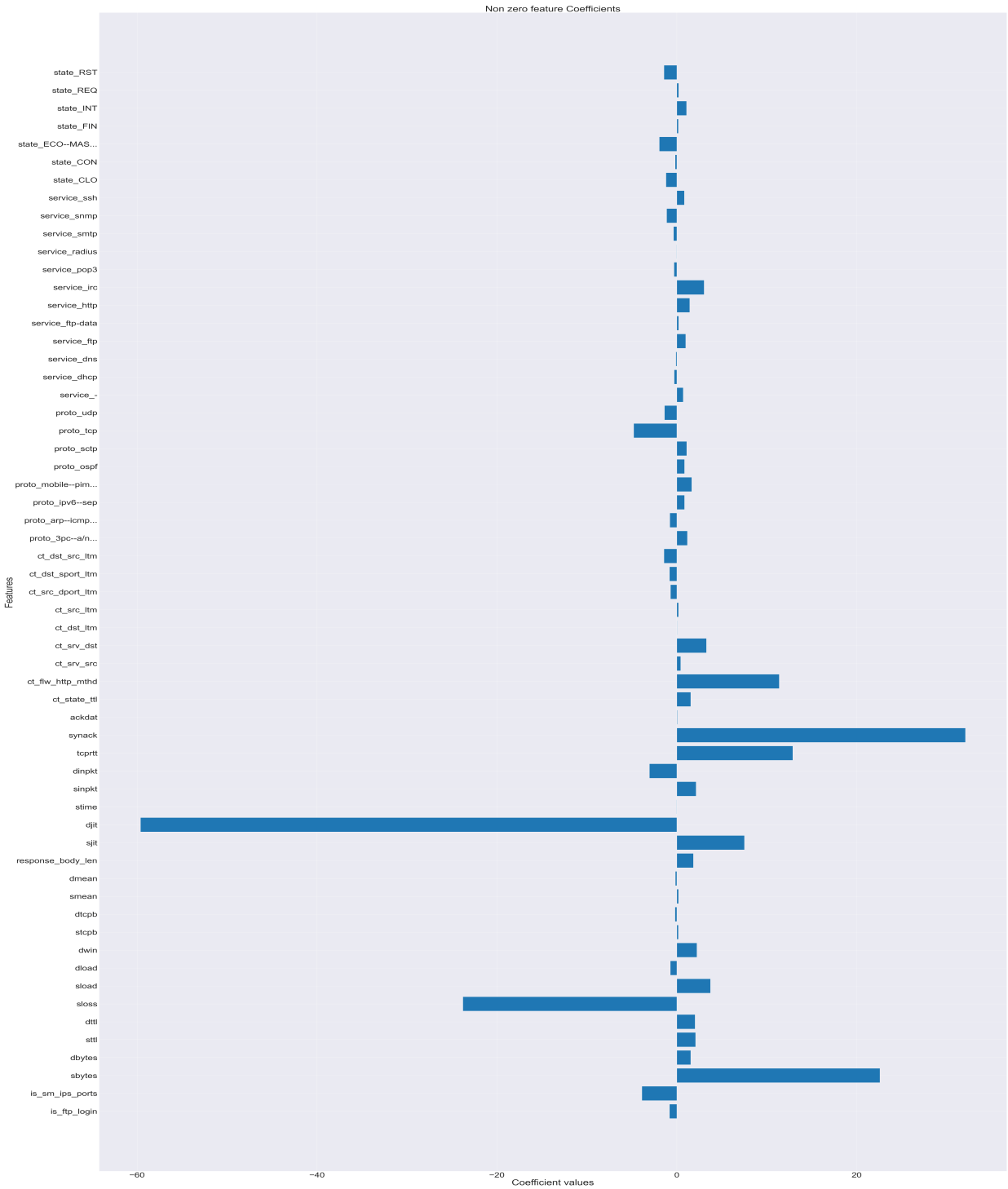

Figure A.2: Coeffcient values using Lasso regularization