Configuration Manual

MSc Research Project
Cybersecurity

Raj Bharath Murali
Student ID: X22240888

School of Computing
National College of Ireland

Supervisor: Michael Pantridge

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Raj Bharath Murali		
Student ID:	X22240888		
Programme:	MSc in Cybersecurity	**Year:**	2023-2024
Module:	MSc. Practicum		
Lecturer:	Michael Pantridge		
Submission Due Date:	12/08/2024		
Project Title:	Optimizing Real-Time DDOS detection with Autoencoders for Enhanced Cybersecurity		
Word Count:	**1913**	**Page Count:**	**17**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Raj Bharath Murali

Date: 11th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Raj Bharath Murali
X22240888

1 Introduction

The proposed research-based solution is designed by incorporating multiple algorithms of deep learning such as (CNN, RNN, Autoencoder) and determining the best performing model which produces higher Accuracy, Precision, Recall, F1-Score in identifying the Real -time DDoS attacks. We have utilized the datasets which containing several DDoS attack types such as Reflection and Exploitation based DDoS attacks with multiple sub divisions of attacks in the form of (TCP, UDP, LDAP, NTP, UDP-Lag etc..). It is crucial to conduct the pre-processing, Feature engineering to ensure the provided data is fair and balanced before training the model. This configuration manual provides comprehensive information on setting up the environment to build the model and test the proposed solution.

2 Requirements

2.1 Machine Configurations

The proposed solution uses different algorithms in the essence of “Deep Learning” which necessities a faster performing machine that helps in executing various libraries for processing the huge data and train the best performing model. Considering, the facts, we have used the below machine to accomplish the proposed solution.

Processor	Intel i7-1065G7 @ 1.30GHz
RAM	16GB-DDR4
Storage	1TB

2.2 Software’s & Tools

The below are the software’s used during the model performance, processing and training the data.

Operating System	Windows 11, Professional, 64-Bit
Python	3.11, 64bit
Anaconda Navigator	2.6.1
Jupyter Notebook Server	6.5.4
Flask	2.2.2

Along with this, we have used HTML/CSS/JavaScript for Web UI and Jinja2 for rendering the web dashboard.

3 Dataset

The dataset used in this proposal helps in training the model to identify the targeted columns to predict the DDoS attack type. We have downloaded the dataset from the following site https://www.unb.ca/cic/datasets/ddos-2019.html. It also contains real-world traffic data, such as PCAP files and CSVs with labeled network flows, targeting various modern DDoS attacks.

Since, the selected raw dataset contains a large number of columns as well as rows which will demand high resources for analytics, model building, compiling and it's training. So, we have pre-processed the data using libraries and eliminate the unnecessary rows, columns and duplicates using "drop" function.

	Protocol	Flow Dura	Total Fwd	Total Back	Fwd Packe	Bwd Packe	Fwd Packe	Fwd Packe	Fwd Packe	Fwd Packe	Bwd Packe	Bwd Packe	Bwd Packe	Bwd Packe	Flow Byte	Flow Pack	Flow IAT M	Flow IAT S	Fl
203	17	2997791	4	0	2064	0	516	516	516	0	0	0	0	0	688.507	1.334316	999263.7	1730774	2
2069	17	107050	4	0	1438	0	389	330	359.5	34.06367	0	0	0	0	13432.98	37.36572	35683.33	61803.61	
401	17	3003373	4	0	2064	0	516	516	516	0	0	0	0	0	687.2274	1.331836	1001124	1733997	3
2612	17	1074	4	0	5888	0	1472	1472	1472	0	0	0	0	0	5482309	3724.395	358	617.4763	
1646	17	1101	42	0	18480	0	440	440	440	0	0	0	0	0	16784742	38147.14	26.85366	98.24575	
1101	17	215109	6	0	2088	0	393	321	348	35.08846	0	0	0	0	9706.707	27.89284	43021.8	58931.55	
463	17	21593	200	0	88000	0	440	440	440	0	0	0	0	0	4075395	9262.261	108.5075	111.5547	
2037	6	98592980	14	0	0	0	0	0	0	0	0	0	0	0	0	0.141998	7584076	8786368	21
2546	17	244	2	0	2928	0	1464	1464	1464	0	0	0	0	0	12000000	8196.722	244	0	
2655	17	106810	4	0	1438	0	389	330	359.5	34.06367	0	0	0	0	13463.16	37.44968	35603.33	61662.45	
2062	17	215702	6	0	2088	0	393	321	348	35.08846	0	0	0	0	9680.021	27.81615	43140.4	59088.7	
994	17	46	2	0	2944	0	1472	1472	1472	0	0	0	0	0	64000000	43478.26	46	0	
2664	6	25835893	4	0	0	0	0	0	0	0	0	0	0	0	0	0.154823	8611964	14916358	25
2475	17	105652	4	0	1398	0	369	330	349.5	22.51666	0	0	0	0	13232.12	37.86015	35217.33	60996.48	
331	6	28411880	6	2	0	0	0	0	0	0	0	0	0	0	0	0.281572	4058840	7165276	17
2391	6	93204975	12	2	0	0	0	0	0	0	0	0	0	0	0	0.150207	7169614	10002280	28
1305	17	5999988	6	0	3096	0	516	516	516	0	0	0	0	0	516.001	1.000002	1199998	1643163	3
868	17	107146	4	0	1438	0	389	330	359.5	34.06367	0	0	0	0	13420.94	37.33224	35715.33	61859.04	
1667	17	45009859	4	0	916	0	229	229	229	0	0	0	0	0	20.3511	0.088869	15003286	25986452	45

Fig.1 Raw dataset

Avg Fwd S	Avg Bwd S	Subflow F	Subflow F	Subflow B	Subflow B	Init Fwd W	Init Bwd W	Fwd Act D	Fwd Seg Si	Active Me	Active Std	Active Ma	Active Mir	Idle Mean	Idle Std	Idle Max	Idle Min	Label
516	0	4	2064	0	0	-1	-1	3	8	0	0	0	0	0	0	0	0	TFTP
359.5	0	4	1438	0	0	-1	-1	3	-1	0	0	0	0	0	0	0	0	UDP-lag
516	0	4	2064	0	0	-1	-1	3	0	0	0	0	0	0	0	0	0	TFTP
1472	0	4	5888	0	0	-1	-1	3	14	0	0	0	0	0	0	0	0	DrDoS_LDAP
440	0	42	18480	0	0	-1	-1	41	20	0	0	0	0	0	0	0	0	DrDoS_NTP
348	0	6	2088	0	0	-1	-1	5	8	0	0	0	0	0	0	0	0	DrDoS_SSDP
440	0	200	88000	0	0	-1	-1	199	8	0	0	0	0	0	0	0	0	DrDoS_DNS
0	0	14	0	0	0	5840	-1	0	20	1	0	1	1	16432162	3287674	21489516	12780372	Syn
1464	0	2	2928	0	0	-1	-1	1	20	0	0	0	0	0	0	0	0	DrDoS_LDAP
359.5	0	4	1438	0	0	-1	-1	3	0	0	0	0	0	0	0	0	0	UDP-lag
348	0	6	2088	0	0	-1	-1	5	0	0	0	0	0	0	0	0	0	DrDoS_UDP
1472	0	2	2944	0	0	-1	-1	1	844	0	0	0	0	0	0	0	0	DrDoS_DNS
0	0	4	0	0	0	5840	-1	0	20	1	0	1	1	25835892	0	25835892	25835892	UDP-lag
349.5	0	4	1398	0	0	-1	-1	3	20	0	0	0	0	0	0	0	0	DrDoS_UDP
0	0	6	0	2	0	5840	0	0	20	48	0	48	48	14205886	4444317	17348492	11063280	UDP-lag
0	0	12	0	2	0	5840	0	0	20	33.5	39.47573	81	1	18640968	5730673	28405576	13438058	Syn
516	0	6	3096	0	0	-1	-1	5	20	0	0	0	0	0	0	0	0	TFTP
359.5	0	4	1438	0	0	-1	-1	3	383	0	0	0	0	0	0	0	0	DrDoS_UDP
229	0	4	916	0	0	-1	-1	3	0	1	0	1	1	45009856	0	45009856	45009856	DrDoS_NetB

Fig.2 Label column

Protocol	Flow Dura	Total Fwd	Total Back	Fwd Packe	Bwd Packe	Fwd Packe	Fwd Packe	Fwd Packe	Fwd Packe	Bwd Packe	Bwd Packe	Bwd Packe	Bwd Packe	Flow Byte	Flow Pack	Flow IAT M	Flow IAT S	Flow IAT M	Fl
17	200	2	0	2944	0	1472	1472	1472	0	0	0	0	0	14720000	10000	200	0	200	
6	98453617	10	0	0	0	0	0	0	0	0	0	0	0	0	0.101571	10939291	15363707	40825536	
6	93404185	10	0	0	0	0	0	0	0	0	0	0	0	0	0.107062	10378243	13029097	27557926	
17	108753	4	0	1398	0	369	330	349.5	22.51666	0	0	0	0	12854.82	36.78059	36251	62745.28	108703	
17	20555	2	2	84	116	42	42	42	0	58	58	58	0	9729.993	194.5999	6851.667	11863.97	20551	
17	3001950	4	0	2064	0	516	516	516	0	0	0	0	0	687.5531	1.332467	1000650	1733175	3001948	
17	2456789	18	0	9288	0	516	516	516	0	0	0	0	0	3780.544	7.326636	144517	243595.3	852501	
6	66707486	10	0	0	0	0	0	0	0	0	0	0	0	0	0.149908	7411943	9080186	20741562	
17	1	2	0	2848	0	1424	1424	1424	0	0	0	0	0	2.85E+09	2000000	1	0	1	
17	105945	4	0	1398	0	369	330	349.5	22.51666	0	0	0	0	13195.53	37.75544	35315	61165.64	105943	
6	16114012	4	0	0	0	0	0	0	0	0	0	0	0	0	0.248231	5371338	9303428	16114011	
17	49	2	0	2944	0	1472	1472	1472	0	0	0	0	0	60081632	40816.33	49	0	49	
17	25695	150	0	65424	0	440	152	436.16	33.14359	0	0	0	0	2546176	5837.711	172.4497	190.6761	825	
6	895	4	0	123	0	46	0	30.75	21.68525	0	0	0	0	137430.2	4469.274	298.3333	514.9965	893	
17	2	2	0	254	0	127	127	127	0	0	0	0	0	1.27E+08	1000000	2	0	2	
6	1.09E+08	14	2	0	0	0	0	0	0	0	0	0	0	0	0.146796	7266342	11326512	39506300	
17	45363967	4	0	916	0	229	229	229	0	0	0	0	0	20.19224	0.088176	15121322	26190896	45363964	
17	217997	6	0	2142	0	402	330	357	35.08846	0	0	0	0	9825.823	27.52332	43599.4	59703.36	110004	
17	3	4	0	916	0	229	229	229	0	0	0	0	0	3.05E+08	1333333	1	0	1	

Fig.3 Pre-processed data

Avg Fwd S	Avg Bwd S	Subflow F	Subflow F	Subflow B	Subflow B	Init Fwd W	Init Bwd W	Fwd Act D	Fwd Seg Si	Active Me	Active Std	Active Ma	Active Mi	Idle Mean	Idle Std	Idle Max	Idle Min	Label
1472	0	2	2944	0	0	-1	-1	1	1448	0	0	0	0	0	0	0	0	Reflection_T
0	0	10	0	0	0	5840	-1	0	20	12.75	23.5	48	1	24613392	13442279	40825536	13262430	Exploitation_
0	0	10	0	0	0	5840	-1	0	20	1	0	1	1	23351044	6984390	27557926	12939714	Exploitation_
349.5	0	4	1398	0	0	-1	-1	3	0	0	0	0	0	0	0	0	0	Exploitation_
42	58	2	84	2	116	-1	-1	1	20	0	0	0	0	0	0	0	0	Benign
516	0	4	2064	0	0	-1	-1	3	8	0	0	0	0	0	0	0	0	Reflection_U
516	0	18	9288	0	0	-1	-1	17	8	0	0	0	0	0	0	0	0	Reflection_U
0	0	10	0	0	0	5840	-1	0	20	1	0	1	1	16676870	3721927	20741562	13257206	Exploitation_
1424	0	2	2848	0	0	-1	-1	1	1323	0	0	0	0	0	0	0	0	Reflection_T
349.5	0	4	1398	0	0	-1	-1	3	375	0	0	0	0	0	0	0	0	Reflection_T
0	0	4	0	0	0	5840	-1	0	20	1	0	1	1	16114011	0	16114011	16114011	Exploitation_
1472	0	2	2944	0	0	-1	-1	1	333	0	0	0	0	0	0	0	0	Reflection_T
436.16	0	150	65424	0	0	-1	-1	149	14	0	0	0	0	0	0	0	0	Reflection_T
30.75	0	4	123	0	0	255	-1	2	20	0	0	0	0	0	0	0	0	Benign
127	0	2	254	0	0	-1	-1	1	8	0	0	0	0	0	0	0	0	Reflection_T
0	0	14	0	2	0	5840	0	0	20	9.5	20.82066	52	1	18165846	11027842	39506300	7492881	Exploitation_
229	0	4	916	0	0	-1	-1	3	-1.1E+09	1	0	1	1	45363964	0	45363964	45363964	Reflection_T
357	0	6	2142	0	0	-1	-1	5	20	0	0	0	0	0	0	0	0	Reflection_T
229	0	4	916	0	0	-1	-1	3	-1.1E+09	0	0	0	0	0	0	0	0	Reflection_T

Fig 4. Identified Labels

4 Implementations

4.1 Libraries Used

We have used the below libraries to process the data and create the balanced dataset before the training of proposed model.

```
import numpy as np                       # importing numpy for numerical, array manipulation
import pandas as pd                      # importing pandas for data manipulation
import sys                               # importing sys library
import matplotlib.pyplot as plt          # importing some basic visualisation libraries
import seaborn as sns
import plotly.graph_objects as go        # importing some advance libraries for visualization
import plotly.io as pio
import plotly.offline as pyo
from plotly.subplots import make_subplots
import plotly.express as px
from sklearn import preprocessing                         # importing preprocessing from sklearn
from sklearn.decomposition import PCA                    # importing PCA for dimension reduction
from sklearn.model_selection import train_test_split    # importing library for data split
from sklearn.preprocessing import MinMaxScaler        # importing min max scalar for data normalisation
from sklearn.preprocessing import LabelEncoder,OneHotEncoder  # importing encoders for data encoding
from collections import Counter                      # importing counter library for counting purpose
from imblearn.over_sampling import SMOTE        # importing smote for oversmapling and data balancing
from imblearn.combine import SMOTETomek

# importing libraries for model building
import tensorflow as tf                                # importing tensorflow
from tensorflow.keras.models import Sequential      # importing different required modules from keras and tensorflow
from tensorflow.keras.layers import Dense, Dropout, Activation,Embedding,Flatten,TimeDistributed,Conv1D,MaxPooling1D,LSTM,Simple
from tensorflow.keras.models import load_model, save_model
from tensorflow.keras.models import Model
import keras
# importing Sklearn for metrics and evaluation
from sklearn.metrics import recall_score,precision_score,roc_auc_score,accuracy_score, f1_score,confusion_matrix
# importing some basic libarary and setting other parameters
import warnings                          # importing warnings
warnings.filterwarnings('ignore')
from plotly.offline import init_notebook_mode      # setting plotly for offline mode
init_notebook_mode(connected=True)
```

Fig.5 Importing Libraries

4.2 Ingesting Raw data file and Pre-processing

We have ingested the raw data file, and targeting the first five rows of data for testing purpose.

Ingesting Raw Data File

```
In [2]: raw_data = pd.read_csv('CIC_DDOS_2019.csv')          # Loading raw data from original csv file

In [3]: raw_data.head()          # getting insights of first five rows of data
Out[3]:
```

	Unnamed: 0	Protocol	Flow Duration	Total Fwd Packets	Total Backward Packets	Fwd Packets Length Total	Bwd Packets Length Total	Fwd Packet Length Max	Fwd Packet Length Min	Fwd Packet Length Mean	...	Fwd Seg Size Min	Active Mean	Active Std	Active Max	Active Min	Idle Mean	Idle Std	Idle Max	Idl Mi
0	203	17	2997791	4	0	2064.0	0.0	516.0	516.0	516.0	...	8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
1	2069	17	107050	4	0	1438.0	0.0	389.0	330.0	359.5	...	-1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
2	401	17	3003373	4	0	2064.0	0.0	516.0	516.0	516.0	...	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
3	2612	17	1074	4	0	5888.0	0.0	1472.0	1472.0	1472.0	...	14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
4	1646	17	1101	42	0	18480.0	0.0	440.0	440.0	440.0	...	20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.

5 rows × 67 columns

Data Preprocessing

```
In [5]: raw_data = raw_data.drop(['Unnamed: 0'],axis=1)          # Dropping unnecesaary column

In [6]: raw_data = raw_data.drop_duplicates()                    # dropping duplicate from the data
        raw_data.shape                                # getting dimension of data after dropping duplicates

Out[6]: (408862, 66)

In [7]: print(raw_data.isnull().sum())                    # checking null values in data
        print("Total missing values : ",sum(list(raw_data.isnull().sum())))

        Protocol                      0
        Flow Duration                 0
        Total Fwd Packets             0
        Total Backward Packets        0
        Fwd Packets Length Total      0
                                     ..
        Idle Mean                     0
        Idle Std                      0
        Idle Max                      0
        Idle Min                      0
        Label                         0
        Length: 66, dtype: int64
        Total missing values :  0
```

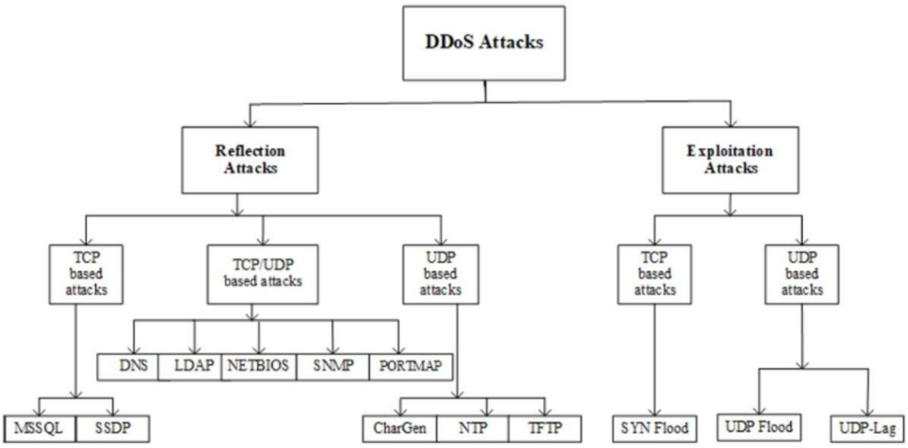
The below screenshot depicts the data type of all columns.

```
Data columns (total 66 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   Protocol                  408862 non-null  int64
 1   Flow Duration             408862 non-null  int64
 2   Total Fwd Packets         408862 non-null  int64
 3   Total Backward Packets    408862 non-null  int64
 4   Fwd Packets Length Total  408862 non-null  float64
 5   Bwd Packets Length Total  408862 non-null  float64
 6   Fwd Packet Length Max     408862 non-null  float64
 7   Fwd Packet Length Min     408862 non-null  float64
 8   Fwd Packet Length Mean    408862 non-null  float64
 9   Fwd Packet Length Std     408862 non-null  float64
 10  Bwd Packet Length Max     408862 non-null  float64
 11  Bwd Packet Length Min     408862 non-null  float64
 12  Bwd Packet Length Mean    408862 non-null  float64
 13  Bwd Packet Length Std     408862 non-null  float64
 14  Flow Bytes/s              408862 non-null  float64
 15  Flow Packets/s            408862 non-null  float64
 16  Flow IAT Mean             408862 non-null  float64
 17  Flow IAT Std              408862 non-null  float64
 18  Flow IAT Max              408862 non-null  float64
 19  Flow IAT Min              408862 non-null  float64
 20  Fwd IAT Total             408862 non-null  float64
```

Once the data pre-processing done, we have gained the insights of data types of all the columns using pandas core data functions. In next step we have gathered the statistical analysis to understand what sort of feature engineering is required which identifies (mean, std, min, max).

	Protocol	Flow Duration	Total Fwd Packets	Total Backward Packets	Fwd Packets Length Total	Bwd Packets Length Total	Fwd Packet Length Max	Fwd Packet Length Min	Fwd Packet Length Mean	Fwd Packet Length Std
count	408862.000000	4.088620e+05	408862.000000	408862.000000	408862.000000	4.088620e+05	408862.000000	408862.000000	408862.000000	408862.000000
mean	14.941526	1.097665e+07	45.071870	0.319604	5780.641657	5.624761e+01	557.317838	531.569341	546.717492	10.332966
std	4.299799	2.761495e+07	1682.760123	3.997862	15670.336738	6.557607e+03	469.056623	478.862545	471.502957	27.191844
min	0.000000	1.000000e+00	1.000000	0.000000	0.000000	0.000000e+00	0.000000	0.000000	0.000000	0.000000
25%	17.000000	4.400000e+01	2.000000	0.000000	880.000000	0.000000e+00	369.000000	229.000000	348.000000	0.000000
50%	17.000000	2.912750e+04	4.000000	0.000000	2064.000000	0.000000e+00	440.000000	411.000000	433.828580	0.000000
75%	17.000000	2.999206e+06	6.000000	0.000000	2896.000000	0.000000e+00	613.000000	606.000000	607.000000	22.516661
max	17.000000	1.200000e+08	100148.000000	1666.000000	206080.000000	3.507830e+06	3421.000000	1472.000000	1908.800049	1246.803955

In the next step, we have identified the target column "Label" is having 13 distinct values which makes difficult for the classification. Hence, we have used mapping function to map the distinct values to superclass.



After the above steps performed, we have saved the preprocessed dataset and reading the modified dataset for further process.

```
raw_data = pd.read_csv('preprocessed_dataset.csv')    # reading new modified dataset
raw_data['Label'].value_counts()                     # Different classes in Label Columns
```

```
Label
Reflection_TCP_UDP    124253
Reflection_TCP         79583
Reflection_UDP         79496
Exploitation_UDP       78212
Exploitation_TCP       39919
Benign                  7213
Name: count, dtype: int64
```

After mapping classes, we are having imbalanced data and the Benign classes is having very less rows while the total records are very large in number, therefore we sample 100k records in a way so that all benign rows get included to preserve them while sampling.

```python
# Separating the BENIGN class from the rest to preserve them
benign_class = raw_data[raw_data['Label'] == 'Benign']
remaining_class = raw_data[raw_data['Label'] != 'Benign']

# Sampling the subset from the remaining data
sampled_remaining_data = remaining_class.sample(n=100000, random_state=5)
final_sampled_data = pd.concat([benign_class, sampled_remaining_data])

# Shuffling data
final_sampled_data = final_sampled_data.sample(frac=1, random_state=10)

# Saving the preprocessed sampled dataset to a new CSV file
#final_sampled_data.to_csv("preprocessed_sampled_dataset.csv", index=False)
```

Fig.6 Removing Benign values for balancing the data

After separating the benign class, sampling the subset from the remaining data to process the final data. In the next step we have executed the data reading function again to finalize the data for future engineering.

On the next step, we have conducted the data analysis exploratory, where we have targeted unique values and column label. Similarly, we have used the same strategy for identifying the below columns vs label.

- ✓ FWD PSH Flag
- ✓ SYN Flag
- ✓ RST Flag
- ✓ ACK Flag
- ✓ URG Flag
- ✓ CWE Flag

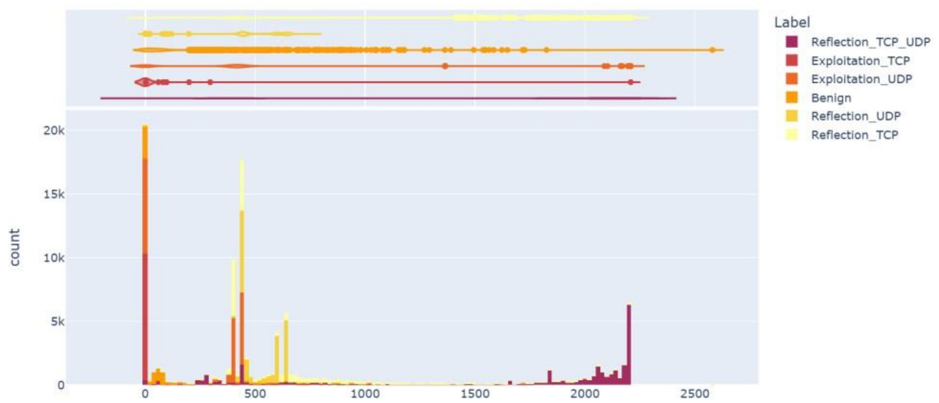
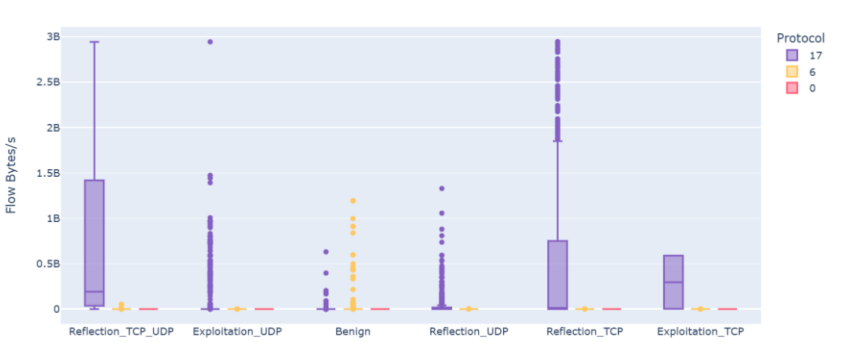


Fig.7 Average packet size

In the next step, we have identified the Fwd_packets and Bwd_Packets, to identify the attacks. We have also conducted the same analysis for bytes.



After, determining the flow packets and bytes, we have compared both the ratios to understand the flow per second. Let's see the comparison.

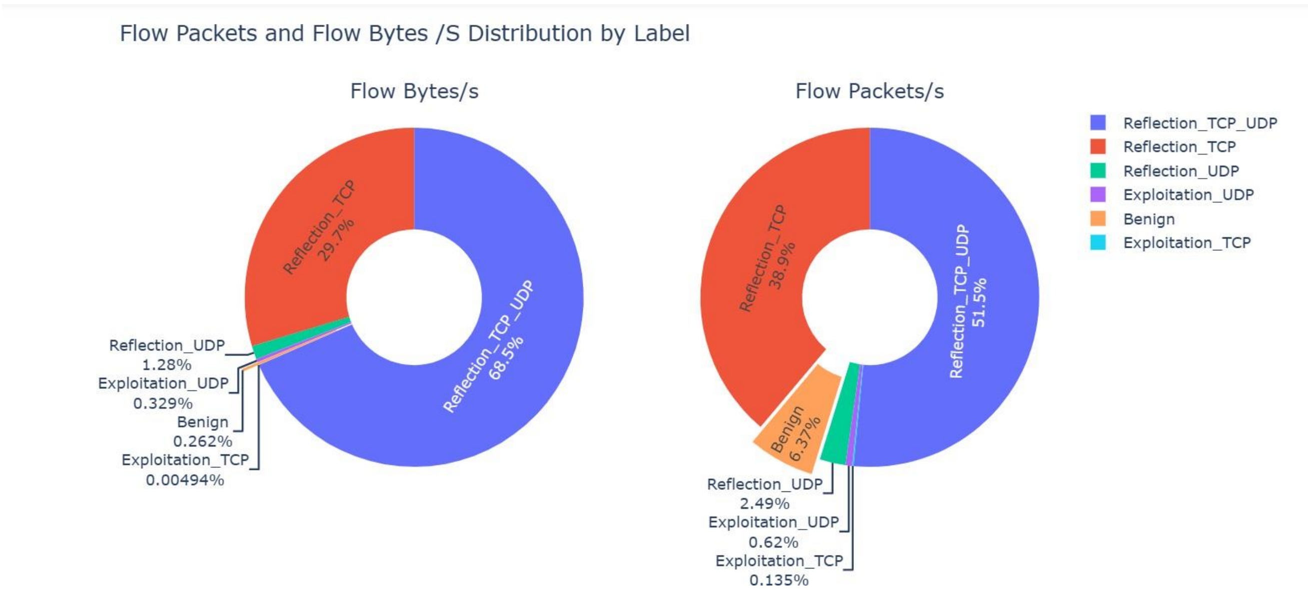


Fig.8 Flow packets and Flow Bytes per second

After determining the flow packets and bytes/S, It highlights the Reflection_TCP_UDP and Reflection_TCP are predominant types of traffic in terms of flow bytes and flow packets per second. Significantly Reflection_TCP_UDP shows the highest traffic, whereas other labels share a smaller portion.

The final part of analysis is to summarize the ratio of all the attacks types. We have used pie chart to determine the contribution of each attack.

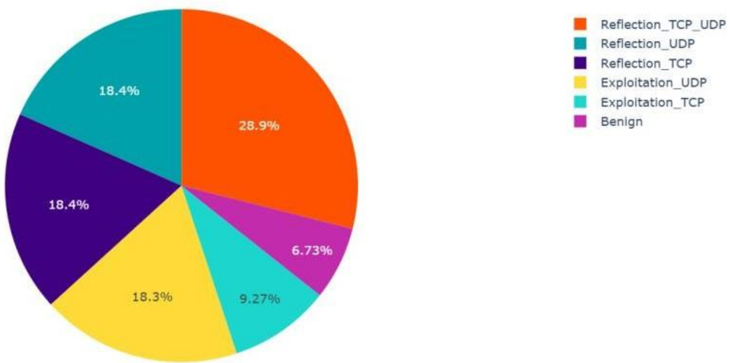


Fig.9 Label Visualization of each attack

Since there is a difference in each label percentages it is considered to be the imbalanced data, which is not suffice to train the model. In order to balance the data, we have done feature engineering by using “Smote” oversampling which balances the data.

```
#dropping target column from data
label_column = final_data['Label']
final_data.drop('Label',inplace=True,axis=1)

sm=SMOTE(random_state=22)                              # balancing imbalanced labels using smote oversampling
extracted_feature, new_label = sm.fit_resample(final_data, label_column)
print("The number of classes before fit {}".format(Counter(label_column)))
print("The number of classes after fit {}".format(Counter(new_label)))

The number of classes before fit Counter({'Reflection_TCP_UDP': 31015, 'Reflection_UDP': 19706, 'Reflection_TCP': 19696, 'Explo
itation_UDP': 19643, 'Exploitation_TCP': 9940, 'Benign': 7213})
The number of classes after fit Counter({'Reflection_TCP_UDP': 31015, 'Exploitation_TCP': 31015, 'Exploitation_UDP': 31015, 'Be
nign': 31015, 'Reflection_UDP': 31015, 'Reflection_TCP': 31015})

true_classes = ['Reflection_TCP_UDP', 'Exploitation_TCP', 'Exploitation_UDP',
       'Benign', 'Reflection_UDP', 'Reflection_TCP']         # visualising the data after data balancing
label_values = new_label.value_counts()
custom_colors = ['#FF5200', '#00A1AB', '#400082', '#FEDB39', '#1CD6CE', '#C32BAD']
px.pie(label_values,names=true_classes,values = label_values,title="Label Classes Visualisation",color_discrete_sequence=custom_c
```

Fig.10 Smote oversampling

After performing smote function, we can see the data is balanced without losing any values.

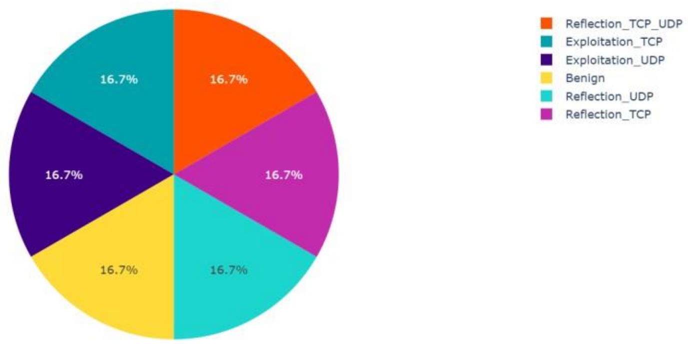


Fig.11 Balanced Label visualization

In the next step, we have performed PCA (**Principal component Analysis**) since the number of columns in this dataset is quite high, which resulted in high dimension of data. In order to reduce the dimension without losing the information and the same can be achieved by implementing unsupervised algorithm PCA.

In the process of PCA, all the 66 columns will convert into components, in which it targets all the data, and we set threshold at 98%. For Instance, if we target 15 columns it will capture 98% of data without losing any accuracy.

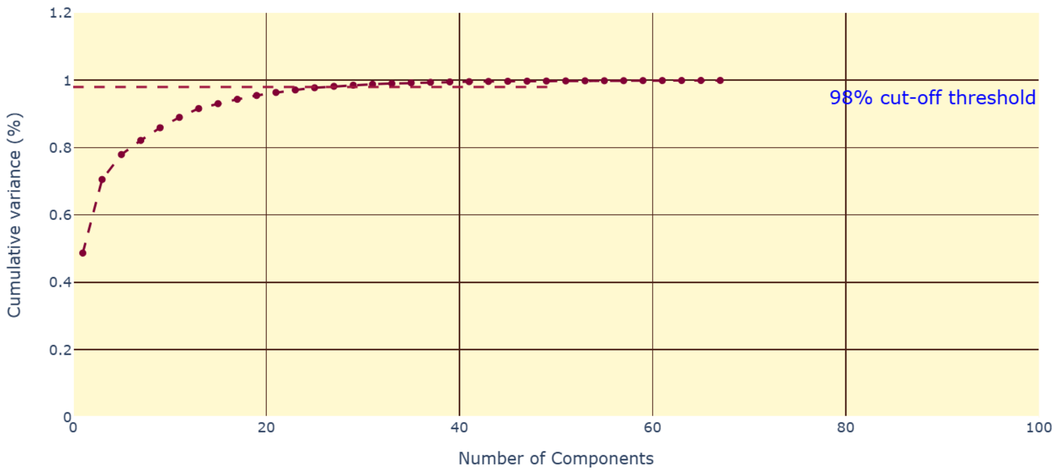


Fig.12 Cumulative Variance

Implementation

In this stage, after balancing the proper data without losing any information. It is crucial to provide the proper data before training the model. We have trained the models using (CNN, RNN, AUTOENCODER), post training of all the models, we would be able to identify the best performing model which gives higher level of (Accuracy, Precision, F1, Recall). Once, determining the best performing model, we will then deploy that model to detect and mitigate the DDoS attacks.

Model Building and Training

In this phase, after having the proper balanced dataset, we will be training the model using deep learning algorithms, and determine the best performing model to detect the real-time modern DDoS attacks. This phase consists several stages, as we will be training model and each algorithm takes time to learn the dataset for future predictions.

```
from sklearn.model_selection import train_test_split  # importing train test split libraray
X_train, X_test, y_train, y_test = train_test_split(pca_ex_df,new_label, test_size =0.3,random_state = 38,shuffle=True)   #splitt
```

```
training_data = np.array(X_train).reshape((-1,1,X_train.shape[1]))    # shaping training data into required format
test_data = np.array(X_test).reshape((-1,1,X_test.shape[1]))      # shaping test data into required format
```

Fig.13 Data shaping

Model 1- Convolutional Neural Network (CNN)

```
# Building CNN Model
model1 =Sequential()                                                          #adding sequential layer
model1.add(tf.keras.layers.Input(shape=(1, X_train.shape[1])))                # adding input layer
model1.add(tf.keras.layers.Conv1D(filters=1, kernel_size=1, activation='linear'))  # adding conv 1d layer
model1.add(tf.keras.layers.Dropout(0.3))                                      # adding dropout layers
model1.add(tf.keras.layers.Flatten())                                         # adding flatten layer
model1.add(tf.keras.layers.Dropout(0.4))                                      # adding dropout layers
model1.add(tf.keras.layers.Dense(24,activation='linear'))                     # adding dense layer
model1.add(tf.keras.layers.Dropout(0.3))                                      # adding dropout layers
model1.add(tf.keras.layers.Dense(6,"softmax"))                                # adding prediction layer
model1.compile(optimizer="adam",loss="categorical_crossentropy",metrics=["accuracy"])   # compiling the model
model1.summary()                                                                        # summary of model
```

Fig 14. Building CNN

```
history1=model1.fit(training_data, y_train, validation_data=(test_data, y_test), batch_size= 64, epochs= 10)   # training CNN moc
```

```
Epoch 1/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.2529 - loss: 1.5968 - val_accuracy: 0.6655 - val_loss: 1.1407
Epoch 2/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.3520 - loss: 1.4443 - val_accuracy: 0.7269 - val_loss: 0.9579
Epoch 3/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.3616 - loss: 1.4087 - val_accuracy: 0.7423 - val_loss: 0.9260
Epoch 4/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.3422 - loss: 1.3975 - val_accuracy: 0.8086 - val_loss: 0.9125
Epoch 5/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.3420 - loss: 1.3939 - val_accuracy: 0.7091 - val_loss: 0.9058
Epoch 6/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.3368 - loss: 1.3951 - val_accuracy: 0.7272 - val_loss: 0.9087
Epoch 7/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.3425 - loss: 1.4014 - val_accuracy: 0.8300 - val_loss: 0.8902
Epoch 8/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.3314 - loss: 1.4000 - val_accuracy: 0.7692 - val_loss: 0.8999
Epoch 9/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.3398 - loss: 1.3922 - val_accuracy: 0.6889 - val_loss: 0.9160
Epoch 10/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.3449 - loss: 1.4010 - val_accuracy: 0.7279 - val_loss: 0.8836
```

Fig.15 Training CNN

After building and training the CNN model, we have got the below parameters in which Accuracy, Precision, Recall and F1_Score identified.

```
# Calculating the accuracy, precision, recall and F1-score of CNN model on test data
accuracy1=accuracy_score(test_classes,y_pred1)
precision1 = precision_score(test_classes, y_pred1, average = 'weighted')
recall1 = recall_score(test_classes, y_pred1, average = 'weighted')
f1_score1 = f1_score(test_classes, y_pred1, average = 'weighted')
print("CNN Model Accuracy: %.2f%%" % (accuracy1*100))
print("CNN Model Precision: %.4f" % (precision1))
print("CNN Model Recall: %.4f" % (recall1))
print("CNN Model F1_score: %.4f" % (f1_score1))
```

```
CNN Model Accuracy: 72.79%
CNN Model Precision: 0.6835
CNN Model Recall: 0.7279
CNN Model F1 score: 0.7021
```

Based on this, we will now proceed with another model.

Model 2 – Recurrent Neural Networks (Rnns)

```
#Building RNN model
model2=Sequential()                                          # adding sequential layer
model2.add(tf.keras.layers.Input(shape=(1, X_train.shape[1])))   # adding input layer
model2.add(tf.keras.layers.BatchNormalization())             # adding batch normalization layer
model2.add(SimpleRNN(5, return_sequences = True))                                 # Adding RNN layer
model2.add(Dropout(0.5))                                   # adding dropout layer
model2.add(SimpleRNN(2))                                       # Adding RNN layer
# model2.add(Dense(30,activation='relu'))                    # adding dense layer
# model2.add(Dropout(0.5))                                     # adding dropout layer
model2.add(Dense(6,activation='linear'))                    # adding dense layer
model2.add(Dropout(0.4))                                     # adding dropout layer
model2.add(Dense(6,activation="softmax"))                    # adding output layer
model2.compile(loss="categorical_crossentropy",optimizer="adam",metrics=["accuracy"])  # compiling model
model2.summary()                                                                     # Summary of model
```

Fig.16 Building RNN

```
history2=model2.fit(training_data, y_train, validation_data=(test_data, y_test), batch_size= 64, epochs= 10, shuffle = Tr

Epoch 1/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - accuracy: 0.2914 - loss: 1.5715 - val_accuracy: 0.5323 - val_loss: 0.7780
Epoch 2/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.4834 - loss: 0.9936 - val_accuracy: 0.7713 - val_loss: 0.6477
Epoch 3/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.5256 - loss: 0.9269 - val_accuracy: 0.7941 - val_loss: 0.5910
Epoch 4/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.5542 - loss: 0.8926 - val_accuracy: 0.8220 - val_loss: 0.5533
Epoch 5/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.5667 - loss: 0.8747 - val_accuracy: 0.9564 - val_loss: 0.5406
Epoch 6/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.5810 - loss: 0.8631 - val_accuracy: 0.9874 - val_loss: 0.5200
Epoch 7/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.5926 - loss: 0.8575 - val_accuracy: 0.8999 - val_loss: 0.4938
Epoch 8/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6162 - loss: 0.8385 - val_accuracy: 0.9642 - val_loss: 0.4129
Epoch 9/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6392 - loss: 0.8156 - val_accuracy: 0.9710 - val_loss: 0.3871
Epoch 10/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6478 - loss: 0.7957 - val_accuracy: 0.9437 - val_loss: 0.3731
```

Fig.17 Training RNN

After successful training of RNN model, we have identified the below parameters which is performing much better than CNN.

```
# Calculating the accuracy, precision, recall and F1-score of RNN model on test data
accuracy2=accuracy_score(test_classes,y_pred2)
precision2 = precision_score(test_classes, y_pred2, average = 'weighted')
recall2 = recall_score(test_classes, y_pred2, average = 'weighted')
f1_score2 = f1_score(test_classes, y_pred2, average = 'weighted')
print("RNN Model Accuracy: %.2f%%" % (accuracy2*100))
print("RNN Model Precision: %.4f" % (precision2))
print("RNN Model Recall: %.4f" % (recall2))
print("RNN Model F1_score: %.4f" % (f1_score2))
```

```
RNN Model Accuracy: 94.37%
RNN Model Precision: 0.9463
RNN Model Recall: 0.9437
RNN Model F1_score: 0.9432
```

As comparing with CNN, RNN is performing better, and producing high accuracy. It is efficient in identifying the attacks. However, we will be analyzing the Autoencoder as next model and comparing the difference in various parameters.

Model 3- Autoencoder (Best performing model**)**

This is our final model, which is an outstanding performing model than other models. Also, it considered to be the most effective model in detecting anomalies, due it's specialized functions (decoder and encoder). It also helps in reducing the noisy data using loss functions.

```
#Building AutoEncoder model
n_features = X_train.shape[1]
model3 = Sequential()
model3.add(tf.keras.layers.Input(shape=(1, n_features)))
# Layer 1 (Encoder)
model3.add(tf.keras.layers.Conv1D(filters=n_features*8, kernel_size=1, activation='relu'))
model3.add(Dropout(0.4))
# Layer 2 (Encoder)
model3.add(tf.keras.layers.Conv1D(filters=n_features*2, kernel_size=1, activation='relu'))
model3.add(Dropout(0.3))
# Layer 3 (Encoder)
model3.add(tf.keras.layers.Conv1D(filters=n_features, kernel_size=1, activation='relu'))
model3.add(Dropout(0.2))

# BottleNeck of autoencoder
model3.add(tf.keras.layers.BatchNormalization())
model3.add(tf.keras.layers.Flatten())
model3.add(tf.keras.layers.Dense(n_features,activation='relu'))
model3.add(tf.keras.layers.Reshape((1,n_features)))

# Layer 1 (Decoder)
model3.add(tf.keras.layers.Conv1DTranspose(filters=n_features,kernel_size=1, activation='relu'))
model3.add(Dropout(0.2))
# Layer 2 (Decoder)
```

Fig.18 Building Autoencoder

```
Epoch 1/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 15s 6ms/step - accuracy: 0.6929 - loss: 0.7367 - val_accuracy: 0.9980 - val_loss: 0.0157
Epoch 2/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 13s 6ms/step - accuracy: 0.9678 - loss: 0.1168 - val_accuracy: 0.9983 - val_loss: 0.0106
Epoch 3/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9794 - loss: 0.0783 - val_accuracy: 0.9988 - val_loss: 0.0071
Epoch 4/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9848 - loss: 0.0618 - val_accuracy: 0.9990 - val_loss: 0.0061
Epoch 5/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9875 - loss: 0.0541 - val_accuracy: 0.9998 - val_loss: 0.0011
Epoch 6/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9894 - loss: 0.0439 - val_accuracy: 0.9999 - val_loss: 0.0020
Epoch 7/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9907 - loss: 0.0383 - val_accuracy: 0.9990 - val_loss: 0.0073
Epoch 8/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.9914 - loss: 0.0371 - val_accuracy: 0.9998 - val_loss: 0.0024
Epoch 9/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9914 - loss: 0.0377 - val_accuracy: 0.9998 - val_loss: 0.0017
Epoch 10/10
2036/2036 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9926 - loss: 0.0329 - val_accuracy: 0.9998 - val_loss: 0.0011
```

Fig.19 Autoencoder Training

```
# Calculating the accuracy, precision, recall and F1-score of Autoencoder model on test data
accuracy3=accuracy_score(test_classes,y_pred3)
precision3 = precision_score(test_classes, y_pred3, average = 'weighted')
recall3 = recall_score(test_classes, y_pred3, average = 'weighted')
f1_score3 = f1_score(test_classes, y_pred3, average = 'weighted')
print("Autoencoder Model Accuracy: %.2f%%" % (accuracy3*100))
print("Autoencoder Model Precision: %.4f" % (precision3))
print("Autoencoder Model Recall: %.4f" % (recall3))
print("Autoencoder Model F1_score: %.4f" % (f1_score3))
```

```
Autoencoder Model Accuracy: 99.98%
Autoencoder Model Precision: 0.9998
Autoencoder Model Recall: 0.9998
Autoencoder Model F1_score: 0.9998
```

Now, we are in the final phase of deploying the model to detect DDos attacks. Before proceeding to the next step, we have compared all the models together to have the better visualization which helps in choosing the model effectively.

Model Comparison

```
result_data = {'Accuracy':[accuracy1,accuracy2,accuracy3],        # creating data contains metrics of evaluation
        'Precision':[precision1,precision2,precision3],
        'Recall':[recall1,recall2,recall3],
        'F1_score':[f1_score1,f1_score2,f1_score3],
        'Algo':['CNN','RNN','Autoencoder'] }
result_data = pd.DataFrame(result_data)        # making dataframe of result data
```

Fig.20 Comparison of Model

	Accuracy	Precision	Recall	F1_score	Algo
0	0.727874	0.683459	0.727874	0.702116	CNN
1	0.943665	0.946292	0.943665	0.943238	RNN
2	0.999839	0.999839	0.999839	0.999839	Autoencoder

Finally, we have determined the best performing model is "Autoencoder" as it's ranging higher in all level of metrics which is quite effective for the proposed solution and necessitates in detecting and mitigating real-time DDoS attack.

Web Implementation

In the next part, we will be following the below steps for implementing the Web UI for testing the attacks. Below are the technologies we used for building the Web UI and real-time monitoring dashboard.

- Flask: Creation of web application within python framework
- TensorFlow/Keras: For loading and running the pre-trained deep learning model
- Pandas: pre-processing and data manipulation
- Scikit-Learn: Scaling and Principal component analysis
- HTML/CSS/JavaScript: Web UI
- Jinja2: Rendering the web dashboard

Installations to be followed

Step 1: Install the python using the following link https://www.python.org/downloads/

Step 2: After installing python, we need to install few libraries which is a pre-requisite to execute the Web Ui that we built.
Installation command: -m pip install flask tensorflow pandas numpy sklearn scikit-learn

```
-m pip install flask tensorflow pandas numpy sklearn scikit-learn
```

Step 3: After installation of all the required libraries, we will now run the following command to start the DDOS system which we build to identify the DDos attacks.

Execute the system using the below provided command.

```
Installing collected packages: threadpoolctl, scipy, joblib, scikit-learn
Successfully installed joblib-1.4.2 scikit-learn-1.5.1 scipy-1.14.0 threadpoolctl-3.5.0
PS C:\Users\Rajbharath\OneDrive\Desktop\Assignment\Sem 3\Practicum\Implementation\DDOS_detection_webapplication> python .\app.py
```

Now our detection system is ready to identify the network packets that we send from client, and it analyzes the attack types and mitigate using Autoencoder model.

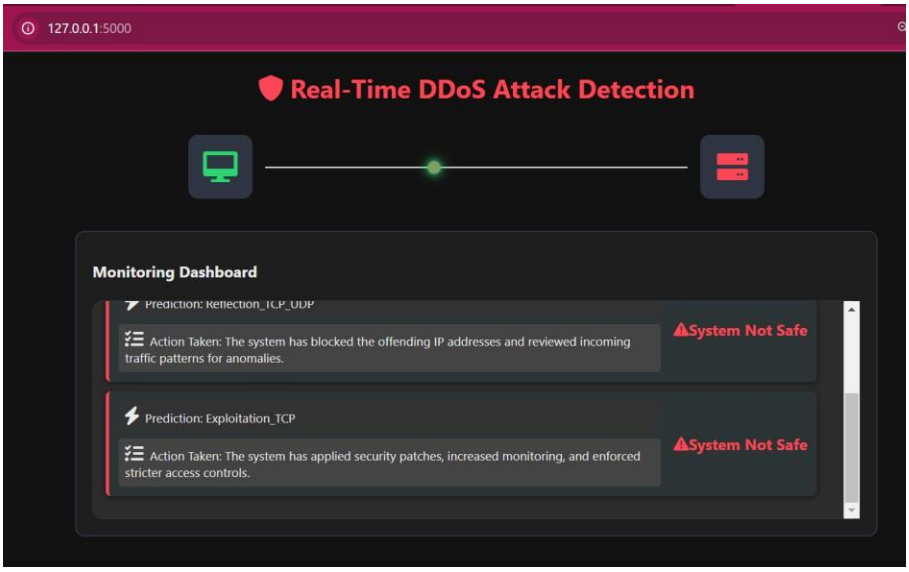
Step 4: Let's start the client, execute the below command to start the client. After successfully running the client, it will start sending the network packets to the server.

```
PS C:\Users\Rajbharath\OneDrive\Desktop\Assignment\Sem 3\Practicum\Implementation\DDOS_detection_webapplication> python
.\client.py
```

After performing all the above steps, we can see the detection system is up and running and analyzing the packets and identifying the attacks.

```
2024-08-08 18:55:00.238007: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly di
fferent numerical results due to floating-point round-off errors from different computation orders. To turn them off, se
t the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-08-08 18:55:02.475535: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly di
fferent numerical results due to floating-point round-off errors from different computation orders. To turn them off, se
t the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-08-08 18:55:10.258705: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to
use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the app
ropriate compiler flags.
C:\Python312\Lib\site-packages\keras\src\optimizers\base_optimizer.py:33: UserWarning: Argument `decay` is no longer sup
ported and will be ignored.
  warnings.warn(
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be e
mpty until you train or evaluate the model.
WARNING:absl:Error in loading the saved optimizer state. As a result, your model is starting with a freshly initialized
optimizer.
 * Serving Flask app 'app'
 * Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI ser
ver instead.
 * Running on http://127.0.0.1:5000
```

Now the server is receiving the packets, based on the dataset and trained model. It is now successfully able to identify the attack types and providing the mitigations.



Note: We have also used time. Sleep function on client, so that it sends packets every 5 seconds and dashboard get auto refresh.