

# Anomaly Detection using Auto ML and DNN

---

20185141 용권순

# Dataset - Anomaly

1.tcp_syn_scan	2017-05-27 오후 5:58	Wireshark capture file	10,478KB
2.tcp_half_open_scan	2017-05-27 오후 7:40	Wireshark capture file	9,343KB
3.fin_scan	2017-05-27 오후 9:08	Wireshark capture file	138KB
4.udp_scan	2017-05-27 오후 11:16	Wireshark capture file	193KB
5.udp_flooding	2016-09-28 오후 8:55	Wireshark capture file	8,445KB

```
fin_scan = pd.read_csv(anomaly_path[0]).drop("No.",axis=1)
tcp_half = pd.read_csv(anomaly_path[1]).drop("No.",axis=1)
tcp_syn = pd.read_csv(anomaly_path[2]).drop("No.",axis=1)
udp_flood = pd.read_csv(anomaly_path[3]).drop("No.",axis=1)
udp_scan = pd.read_csv(anomaly_path[4]).drop("No.",axis=1)
```

각 유형별 CSV파일을 read

```
anomaly_df = pd.concat(anomaly_list, axis=0)
anomaly_df = anomaly_df.reset_index()
anomaly_df = anomaly_df.drop(columns="index", axis=1)
anomaly = anomaly_df.drop("Info", axis=1)
anomaly["class"] = "anomaly"
```

하나의 DataFrame으로 Concat

- Wireshark를 사용해 직접 수집한 패킷을 Dataset을 사용
- Smart lead에 업로드 되어있는 5개의 네트워크 공격 파일을 사용
- Pcap파일을 CSV로 저장한 다음 Pandas로 read

No.	Time	Source	Source Port	Source Port.1	Destination	Destination Port	Destination Port.1	Protocol	Length	Flags	Sequence Number	Acknowledgment Number	Window	Type	Time to Live	Total Length	Code	Length.1
0	1 0.000000	10.40.219.42	NaN	NaN	10.40.201.225	NaN	NaN	ICMP	42	NaN	NaN	NaN	NaN	8.0	43	28	0.0	NaN
1	2 0.000138	10.40.219.42	42325.0	NaN	10.40.201.225	443.0	NaN	TCP	58	0x002	0.0	0.0	1024.0	NaN	47	44	NaN	NaN
2	3 0.000139	10.40.219.42	42325.0	NaN	10.40.201.225	80.0	NaN	TCP	54	0x010	1.0	1.0	1024.0	NaN	38	40	NaN	NaN
3	4 0.000139	10.40.219.42	NaN	NaN	10.40.201.225	NaN	NaN	ICMP	54	NaN	NaN	NaN	NaN	13.0	38	40	0.0	NaN
4	5 0.002188	10.40.201.225	NaN	NaN	10.40.219.42	NaN	NaN	ICMP	60	NaN	NaN	NaN	NaN	0.0	63	28	0.0	NaN

Csv로 읽었을 때

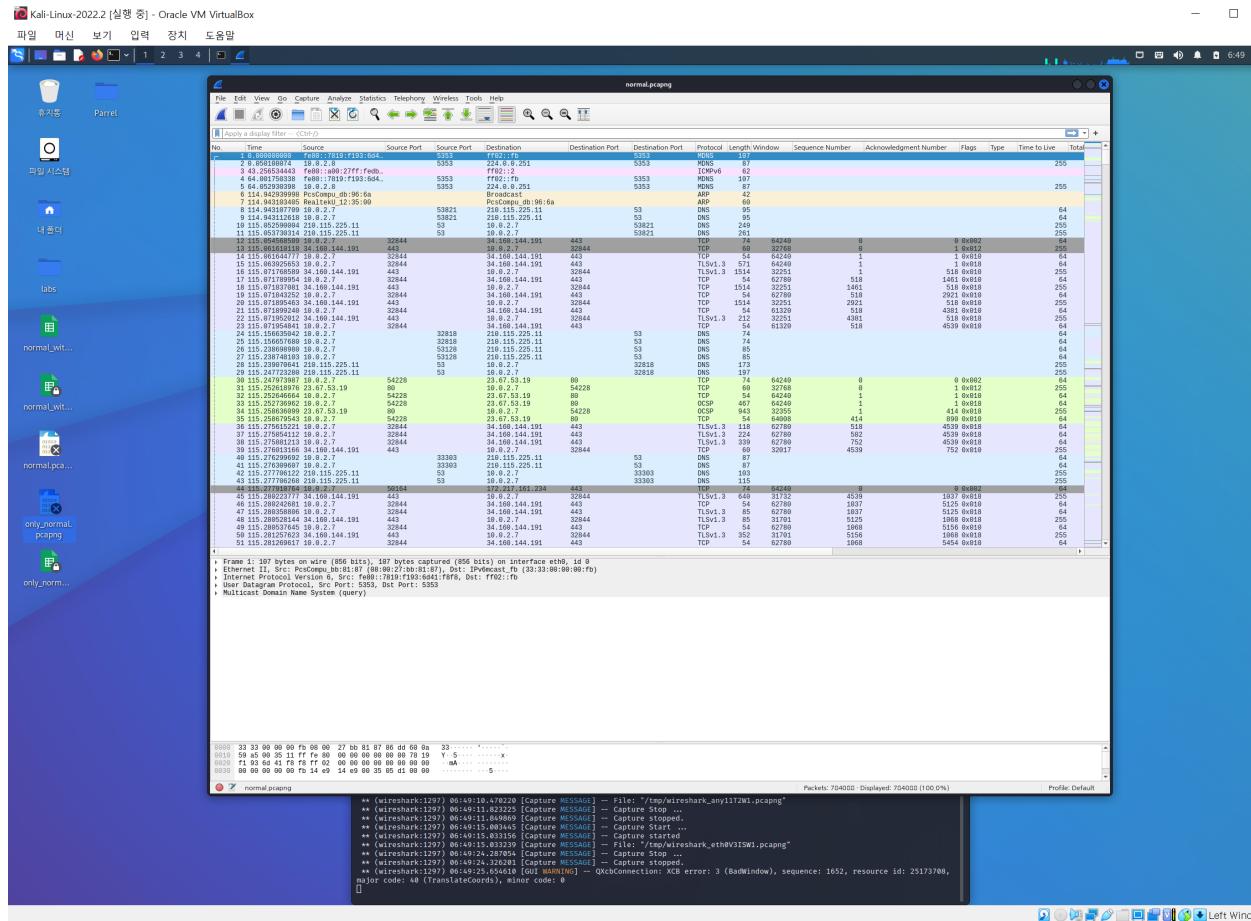
No.	Source	Source Port	Source Port.1	Destination	Destination Port	Destination Port.1	Protocol	Length	Flags	Sequence Number	Acknowledgment Number	Window
1	10.40.219.42	50864		10.40.201.225	80		TCP	78	0x002	0	1	
2	0.00043	10.40.201.225	80	10.40.219.42	443		TCP	78	0x02	0	0	
3	0.00043	10.40.219.42	50865	10.40.219.42	443		TCP	74	0x052	0	1	
4	0.00043	10.40.201.225	443	10.40.219.42	80		TCP	66	0x010	1	1	
5	0.00043	10.40.219.42	50864	10.40.201.225	443		TCP	65	0x010	1	1	
6	0.00043	10.40.219.42	50865	10.40.201.225	443		TCP	60	0x014	1	1	
7	0.00041	10.40.219.42	50864	10.40.201.225	80		TCP	60	0x014	1	1	
8	0.00411	10.40.219.42	50865	10.40.201.225	443		TCP	60	0x014	1	1	
9	0.00537	10.40.219.42	50866	10.40.201.225	21		TCP	60	0x002	0	0	
10	0.00537	10.40.219.42	50866	10.40.201.225	135		TCP	54	0x002	0	0	
11	0.00537	10.40.219.42	50867	10.40.201.225	135		TCP	78	0x02	0	0	
12	0.006782	10.40.201.225	143	10.40.219.42	50867		TCP	78	0x014	1	1	
13	0.006789	10.40.219.42	50869	10.40.201.225	80		TCP	78	0x002	0	0	
14	0.006795	10.40.201.225	5087	10.40.219.42	50869		TCP	54	0x014	1	1	
15	0.006795	10.40.219.42	5087	10.40.201.225	21		TCP	78	0x002	0	0	
16	0.006797	10.40.201.225	5087	10.40.219.42	50868		TCP	54	0x014	1	1	
17	0.007048	10.40.219.42	50870	10.40.201.225	135		TCP	60	0x002	0	0	
18	0.007052	10.40.201.225	135	10.40.219.42	50870		TCP	54	0x014	1	1	
19	0.007151	10.40.219.42	50871	10.40.201.225	113		TCP	78	0x002	0	0	
20	0.007152	10.40.219.42	50871	10.40.201.225	100		TCP	54	0x014	1	1	
21	0.007152	10.40.219.42	50872	10.40.201.225	203		TCP	78	0x02	0	0	
22	0.007215	10.40.201.225	993	10.40.219.42	50872		TCP	54	0x014	1	1	
23	0.007305	10.40.219.42	50874	10.40.201.225	554		TCP	78	0x02	0	0	
24	0.007312	10.40.201.225	554	10.40.219.42	50874		TCP	54	0x014	1	1	
25	0.007312	10.40.219.42	50875	10.40.201.225	100		TCP	78	0x02	0	0	
26	0.007381	10.40.201.225	1015	10.40.219.42	50875		TCP	54	0x014	1	1	
27	0.008842	10.40.219.42	50876	10.40.201.225	139		TCP	78	0x002	0	0	
28	0.008856	10.40.201.225	139	10.40.219.42	50876		TCP	54	0x014	1	1	
29	0.009529	10.40.219.42	50877	10.40.201.225	500		TCP	78	0x002	0	0	
30	0.011136	10.40.219.42	50877	10.40.201.225	500		TCP	54	0x014	1	1	
31	0.011136	10.40.219.42	50878	10.40.201.225	3300		TCP	78	0x002	0	0	
32	0.011136	10.40.201.225	3300	10.40.219.42	50878		TCP	74	0x012	0	1	
33	0.011136	10.40.219.42	50879	10.40.201.225	110		TCP	78	0x002	0	0	

Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)  
> Ethernet II, Src: Juniper\_NF\_10\_10 (00:10:42:ff:18:01), Dst: Dell\_1 (00:26:b0:48:ff:97) (00:26:b0:48:ff:97)  
> Internet Protocol Version 4, Src: 10.40.219.42, Dst: 10.40.201.225  
> Transmission Control Protocol, Src Port: 50864, Dst Port: 80, Seq: 0, Len: 0

Packets: 130846 - Displayed: 130846 (100.0%) | Profile: Default

Pcap파일

# Dataset - Normal



	Time	Source	Source Port	Source Port.1	Destination	Destination Port	Destination Port.1	Protocol	Length	Window	Sequence Number	Acknowledgment Number	Flags	Type	Time to Live	Total Length
0	0.000000	PcsCompu_db:96:6a	NaN	NaN	Broadcast	NaN	NaN	ARP	42	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	0.000203	RealtekU_12:35:00	NaN	NaN	PcsCompu_db:96:6a	NaN	NaN	ARP	60	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	0.000207	10.0.2.7	NaN	34919.0	210.115.225.11	NaN	53.0	DNS	95	NaN	NaN	NaN	NaN	NaN	64.0	81.0
3	0.000212	10.0.2.7	NaN	34919.0	210.115.225.11	NaN	53.0	DNS	95	NaN	NaN	NaN	NaN	NaN	64.0	81.0
4	0.106110	210.115.225.11	NaN	53.0	10.0.2.7	34919.0	210.115.225.11	DNS	249	NaN	NaN	NaN	NaN	NaN	255.0	235.0
124208	1352.250301	10.0.2.7	47984.0	NaN	27.0.237.155	443.0	NaN	TCP	54	65535.0	12593.0	2562519.0	0x010	NaN	64.0	40.0
124209	1354.548231	10.0.2.7	51364.0	NaN	172.217.25.163	80.0	NaN	TCP	54	63791.0	833.0	1404.0	0x010	NaN	64.0	40.0
124210	1354.548366	172.217.25.163	80.0	NaN	10.0.2.7	51364.0	NaN	TCP	60	31935.0	1404.0	834.0	0x010	NaN	255.0	40.0
124211	1358.644182	10.0.2.7	52436.0	NaN	117.18.237.29	80.0	NaN	TCP	54	63920.0	414.0	800.0	0x010	NaN	64.0	40.0
124212	1358.644334	117.18.237.29	80.0	NaN	10.0.2.7	52436.0	NaN	TCP	60	32354.0	800.0	415.0	0x010	NaN	255.0	40.0

Normal 유저의 Data packet

- 정상 유저의 Network를 확인하기 위해서 kali linux에서 wireshark를 사용하여 정상적인 유저의 traffic을 캡쳐(ex>Youtube, Naver, Hallym Univ site 방문 등)

# EDA(Exploratory Data Analysis)

No.	Time	Source	Source Port	Source Port.1	Destination	Destination Port	Destination Port.1	Protocol	Length	Flags	Sequence Number	Acknowledgment Number	Window	Type	Time to Live	Total Length	Code	Length.1	Info
0	1 0.000000	10.40.219.42	Nan	NaN	10.40.201.225	Nan	Nan	ICMP	42	NaN	NaN	NaN	NaN	8.0	43	28	0.0	NaN	Echo (ping) request id=0x94de, seq=0/0, ttl=4...
1	2 0.000138	10.40.219.42	42325.0	NaN	10.40.201.225	443.0	Nan	TCP	58	0x002	0.0	0.0	1024.0	NaN	47	44	NaN	NaN	42325 > 443 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
2	3 0.000139	10.40.219.42	42325.0	NaN	10.40.201.225	80.0	Nan	TCP	54	0x010	1.0	1.0	1024.0	NaN	38	40	NaN	NaN	42325 > 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
3	4 0.000139	10.40.219.42	Nan	NaN	10.40.201.225	Nan	Nan	ICMP	54	NaN	NaN	NaN	NaN	13.0	38	40	0.0	NaN	Timestamp request id=0x8d05, seq=0/0, ttl=38
4	5 0.002188	10.40.201.225	Nan	NaN	10.40.219.42	Nan	Nan	ICMP	60	NaN	NaN	NaN	NaN	0.0	63	28	0.0	NaN	Echo (ping) reply id=0x94de, seq=0/0, ttl=6...

- 기본적으로 20개의 Feature가 존재함, 해당하는 protocol에 맞는 feature가 아니면 Nan값으로 되어있음. 전처리해야하는 column이 많이 존재

```
fin_scan.columns , len(fin_scan.columns)

(Index(['Time', 'Source', 'Source Port', 'Source Port.1', 'Destination',
        'Destination Port', 'Destination Port.1', 'Protocol', 'Length', 'Flags',
        'Sequence Number', 'Acknowledgment Number', 'Window', 'Type',
        'Time to Live', 'Total Length', 'Code', 'Length.1', 'Info'],
      dtype='object'),
 19)
```

19개의 feature (No.는 read할때 drop)

# Preprocessing – Port ver.

```
def port_device(dataFrame):
    dataFrame["Source_Port"] = np.round(dataFrame["Source_Port"].fillna(-0.4)+ dataFrame["Source_Port.1"].fillna(-0.4)) #nan값 0으로 채우기
    dataFrame["Destination_Port"] = np.round(dataFrame["Destination_Port"].fillna(-0.4)+ dataFrame["Destination_Port.1"].fillna(-0.4))
    dataFrame = dataFrame.drop(columns=["Source_Port", "Source_Port.1"],axis=1)
    dataFrame = dataFrame.drop(columns=["Destination_Port", "Destination_Port.1"],axis=1)
    return dataFrame
```

	Source_Port	Destination_Port
0	-1.0	-1.0
1	42325.0	443.0
2	42325.0	80.0
3	-1.0	-1.0
4	-1.0	-1.0
...	...	...
396479	47984.0	443.0
396480	51364.0	80.0
396481	80.0	51364.0
396482	52436.0	80.0
396483	80.0	52436.0

- Wireshark에서 UDP와 TCP의 Port를 다르게 판단하기 때문에 하나로 묶어 주는 작업이 필요
- 여기서 nan값(port를 사용하지 않음 ex)ICMP)은 port에서는 사용하지 않는 -1로 설정하여 사용하지 않음을 표시

Port 전처리

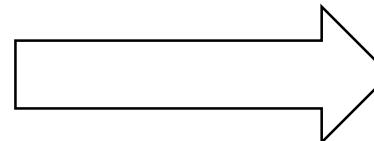
# Preprocessing – Protocol ver.

```
def protocol_onehot(data): #anomaly detection에서 주요하게 쓰이느 Protocol로의 구성
    data["top_protocol"] = "Other"
    top7 = ["TCP", "UDP", "ICMP", "DNS", "TLSv1.2", "TLSv1.3", "QUIC"]
    for i, protocol in enumerate(top7):
        # print(protocol,i)
        data.loc[data["Protocol"].str.contains(protocol), "top_protocol"] = f"{protocol}"

    data = data.drop("Protocol", axis=1)
    protocol_onehot = pd.get_dummies(data["top_protocol"])
    data = pd.concat([data, protocol_onehot], axis=1)
    data.drop("top_protocol", axis=1, inplace=True)
    return data
```

- Anomaly 상황에서 자주 쓰이는 Protocol과 Normal에서 자주 사용되는 Protocol을 one-hot encoding을 진행, 그 외의 Protocol은 탐지에 방해가 될 것 같아서 Other로 통일함
- Ordinary encoding(1,2,3등)을 하지 않는 이유는 숫자로 encoding을 진행할 경우 숫자의 순서를 학습하려는 경향이 있기 때문

0	ICMP
1	TCP
2	TCP
3	ICMP
4	ICMP
	...
272266	ICMP
272267	UDP
272268	ICMP
272269	UDP
272270	ICMP



One hot encoding

	dataset[["TCP", "UDP", "ICMP", "DNS", "TLSv1.2", "TLSv1.3", "QUIC", "Other"]]							
	TCP	UDP	ICMP	DNS	TLSv1.2	TLSv1.3	QUIC	Other
0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0
4	0	0	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...
396479	1	0	0	0	0	0	0	0
396480	1	0	0	0	0	0	0	0
396481	1	0	0	0	0	0	0	0
396482	1	0	0	0	0	0	0	0
396483	1	0	0	0	0	0	0	0

# Preprocessing – Flags ver.

```
def flags_onehot(data):
    onehot = pd.get_dummies(data["Flags"])
    data = data.drop("Flags",axis=1)
    return pd.concat([data,onehot],axis=1)
```

## flags

16비트로 되어있음  
0x014 (RST,ACK)  
0x002 : SYN  
0x0c2 :(SYN, ECE , CWR)  
0x001 : FIN  
0x012 : (SYN,ACK)  
0x010 : ACK  
0x004 : RST  
0x052 :(SYN, ACK, ECN)

- Flags의 종류는 8가지로 TCP로 hand sake를 진행할 때 flags값으로 판별함
- 마찬가지로 one hot encoding을 진행

	0x014	0x002	0x0c2	0x001	0x012	0x010	0x004	0x052
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...
396479	0	0	0	0	0	1	0	0
396480	0	0	0	0	0	1	0	0
396481	0	0	0	0	0	1	0	0
396482	0	0	0	0	0	1	0	0
396483	0	0	0	0	0	1	0	0

One hot encoding을 진행

# Preprocessing – TTL ver.

```
def divide_TTL(data):
    data["RST_TTL"] = -1
    for idx,i in enumerate(data["Time to Live"]):
        s,d = 0,0
        if type(i) ==str:
            if i.find(",")>0:
                s,d = i.split(",")
                data["RST_TTL"][idx] = d
                data["Time to Live"][idx] = s
    return data
```

anomaly[["Time to Live"]]	
Time to Live	
0	43
1	47
2	38
3	38
4	63
...	...
272266	64,49
272267	44
272268	64,44
272269	57
272270	64,57

dataset[["RST_TTL","Time to Live"]]		
RST_TTL	Time to Live	
0	-1	43
1	-1	47
2	-1	38
3	-1	38
4	-1	63
...	...	...
396479	-1	64.0
396480	-1	64.0
396481	-1	255.0
396482	-1	64.0
396483	-1	255.0

2개의 TTL로 divide

Source의 정보와 Destination의 정보가 같이 들어있음

- Flags와 마찬가지로 Three hand shake를 하기 위해서 보내주는 syn정보에 Source와 Destination의 정보가 같이 들어있기에 분리하는 작업이 필요

# Preprocessing – Total Length ver.

```
return data
def divide_Total_length(data):
    data["RST_Total_length"] = -1
    for idx,i in enumerate(data["Total Length"]):
        s,d = 0,0
        if type(i) ==str:
            if i.find(",")>0:
                s,d = i.split(",")
            data["RST_Total_length"][idx] = d
            data["Total Length"][idx] = s
    return data
```

anomaly[["Total Length"]]	
Total Length	
0	28
1	44
2	40
3	40
4	28
...	...
272266	56,28
272267	28
272268	56,28
272269	28
272270	56,28

dataset[["RST_Total_length","Total Length"]]		
RST_Total_length	Total Length	
0	-1	28
1	-1	44
2	-1	40
3	-1	40
4	-1	28
...	...	...
396479	-1	40.0
396480	-1	40.0
396481	-1	40.0
396482	-1	40.0
396483	-1	40.0

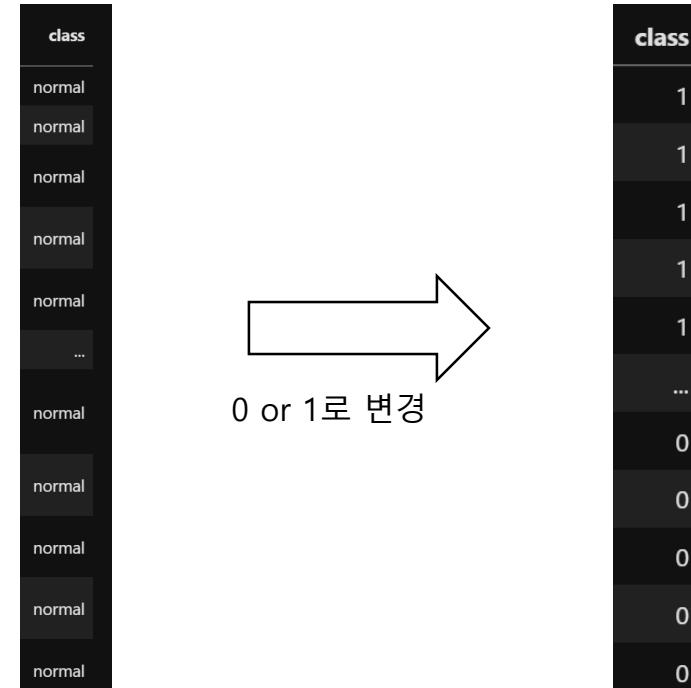
2개의 Total length  
divide

Source의 정보와 Destination의 정보가 같이 들어있음

- 역시 마찬가지로 TCP protocol을 위한 분리 작업을 진행

# Preprocessing – Label ver.

```
def label(data):
    data.loc[data["class"]=="normal","class"] = 0
    data.loc[data["class"]=="anomaly","class"] = 1
    return data
```



- Categorical data를 숫자로 처리하기 위해서 0 or 1로 변경

# Dataset preprocessing

```
def preprocessing(dataFrame): #port를 나누는 부분
    dataFrame = port_device(dataFrame)
    dataFrame = protocol_onehot(dataFrame)
    dataFrame = flags_onehot(dataFrame)
    dataFrame = devide_TTL(dataFrame)
    dataFrame = devide_Total_length(dataFrame)
    dataFrame = label(dataFrame)
    dataFrame = dataFrame.fillna(-1)
    dataFrame = dataFrame.drop(["index", "Time", "Source", "Destination", "Info"], axis=1) #학습에 방해가 되는 column을 지웠습니다.
    return dataFrame
```

- 전체적인 전처리 과정은 위의 함수의 순서대로 진행이 된다

```
dataset = pd.concat([anomaly, normal])
dataset = dataset.reset_index()

dataset = preprocessing(dataset)
```

- 정상 데이터와 이상치로만 이루어져있는 데이터를 concat한 다음 전체에 대해서 preprocessing을 진행 (nan값은 모두 -1로 처리)
- Dataset.colum을 보면 알겠지만, index, time, Source, Destination, Info등 학습에 어려움을 줄 것 같은 feature는 drop을 진행
- Time, Source, Destination같은 feature는 학습에 도움을 줄 수 있지만 Dataset자체가 여러 시나리오를 합친 것이라 시간이 중복성이 존재하고, Source와 Destination의 경우 가상환경에서 진행되었기 때문에 공격을 하는 Host와 피해자가 일정하다는 특징이 있어서 drop하기로 결정

```
dataset.columns
Index(['Length', 'Sequence Number', 'Acknowledgment Number', 'Window', 'Type',
       'Time to Live', 'Total Length', 'Code', 'Length.1', 'class',
       'Source_Port', 'Destination_Port', 'DNS', 'ICMP', 'Other', 'QUIC',
       'TCP', 'TLSv1.2', 'TLSv1.3', 'UDP', '0x001', '0x002', '0x004', '0x010',
       '0x011', '0x012', '0x014', '0x018', '0x019', '0x052', '0x0c2',
       'RST_TTL', 'RST_Total_length'],
      dtype='object')
```

총 33개의 column이 생성

# Dataset Visualize

	Length	Sequence Number	Acknowledgment Number	Window	Type	Time to Live	Total Length	Code	Length.1	class	...	0x010	0x011	0x012	0x014	0x018	0x019	0x052	0x0c2	RST_TTL	RST_Total_length
0	42	-1.0	-1.0	-1.0	8.0	43	28	0.0	-1.0	1	...	0	0	0	0	0	0	0	0	-1	-1
1	58	0.0	0.0	1024.0	-1.0	47	44	-1.0	-1.0	1	...	0	0	0	0	0	0	0	0	-1	-1
2	54	1.0	1.0	1024.0	-1.0	38	40	-1.0	-1.0	1	...	1	0	0	0	0	0	0	0	-1	-1
3	54	-1.0	-1.0	-1.0	13.0	38	40	0.0	-1.0	1	...	0	0	0	0	0	0	0	0	-1	-1
4	60	-1.0	-1.0	-1.0	0.0	63	28	0.0	-1.0	1	...	0	0	0	0	0	0	0	0	-1	-1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
396479	54	12593.0	2562519.0	65535.0	-1.0	64.0	40.0	-1.0	-1.0	0	...	1	0	0	0	0	0	0	0	-1	-1
396480	54	833.0	1404.0	63791.0	-1.0	64.0	40.0	-1.0	-1.0	0	...	1	0	0	0	0	0	0	0	-1	-1
396481	60	1404.0	834.0	31935.0	-1.0	255.0	40.0	-1.0	-1.0	0	...	1	0	0	0	0	0	0	0	-1	-1
396482	54	414.0	800.0	63920.0	-1.0	64.0	40.0	-1.0	-1.0	0	...	1	0	0	0	0	0	0	0	-1	-1
396483	60	800.0	415.0	32354.0	-1.0	255.0	40.0	-1.0	-1.0	0	...	1	0	0	0	0	0	0	0	-1	-1

396484 rows x 33 columns



- Dataset에 대해서 Box Plot을 그려보았지만, 유의미한 분석이 되지 않는다
- 이는 Protocol이란 것 자체가 통신을 하기 위한 인위적인 약속이기 때문

# Dataset Visualize 2

```
def visualize_pca(data,label,sample_num=300,normalize=False):
    pca = PCA(n_components=2) # 2 차원으로 축소
    components = pca.fit_transform(data) #data에 대해서 PCA를 진행
    if normalize:
        MEAN = np.mean(components, axis=1).reshape(-1,1)
        STD = np.std(components, axis=1).reshape(-1,1)
        components = (components-MEAN)/STD
    plt.figure(figsize=(6,6))

    normal_x = []
    normal_y = []
    anomaly_x = []
    anomaly_y = []
    for idx,i in enumerate(label):
        if i ==1:
            normal_x.append(components[idx][0])
            normal_y.append(components[idx][1])
        else:
            anomaly_x.append(components[idx][0])
            anomaly_y.append(components[idx][1])
    plt.scatter(normal_x[:sample_num], normal_y[:sample_num], color="red", label="normal")
    plt.scatter(anomaly_x[:sample_num-200], anomaly_y[:sample_num-200], color="blue", label="anomaly")
    plt.legend()
    plt.show()
```

2-dimension으로 변환

```
def visualize_3d(data,label,sample_num=300,normalize=False): # 3d
    pca_3d = PCA(n_components=3)
    components2 = pca_3d.fit_transform(data)
    if normalize:
        MEAN = np.mean(components, axis=1).reshape(-1,1)
        STD = np.std(components, axis=1).reshape(-1,1)
        components = (components-MEAN)/STD
    normal_x = []
    normal_y = []
    normal_z = []
    anomaly_x = []
    anomaly_y = []
    anomaly_z = []

    for idx,i in enumerate(label):
        if i ==1:
            normal_x.append(components2[idx][0])
            normal_y.append(components2[idx][1])
            normal_z.append(components2[idx][2])
        else:
            anomaly_x.append(components2[idx][0])
            anomaly_y.append(components2[idx][1])
            anomaly_z.append(components2[idx][2])

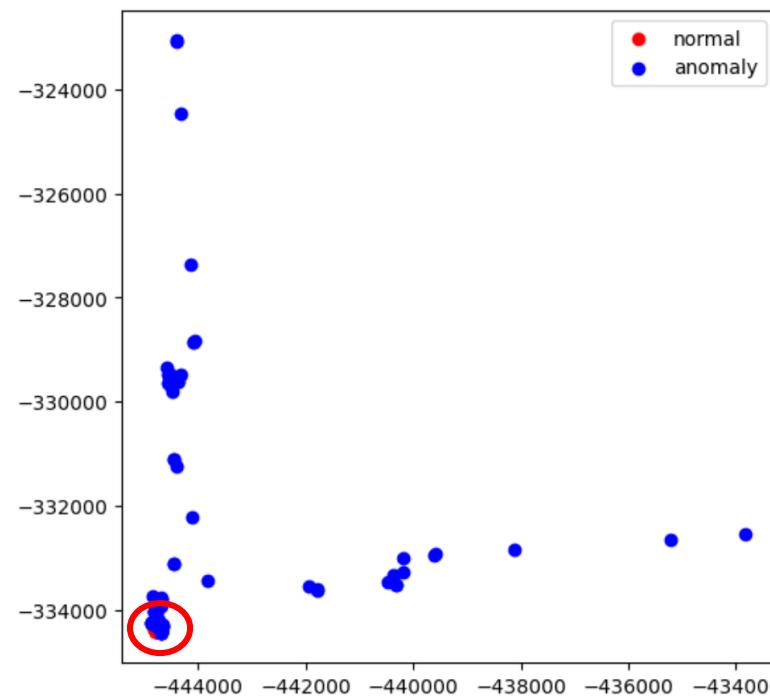
    fig, ax= plt.subplots(1,1, subplot_kw={"projection":"3d"})
    fontlabel = {"fontsize": "large", "color": "gray", "fontweight": "bold"}

    # for idx,ax in enumerate(axes):
    ax.scatter(normal_x[:sample_num], normal_y[:sample_num],normal_z[:sample_num], color="red", label="normal")
    ax.scatter(anomaly_x[:sample_num], anomaly_y[:sample_num],anomaly_z[:sample_num],color="blue", label="anomaly")
    ax.view_init(elev=30,azim=20)
    ax.legend()
    plt.show()
```

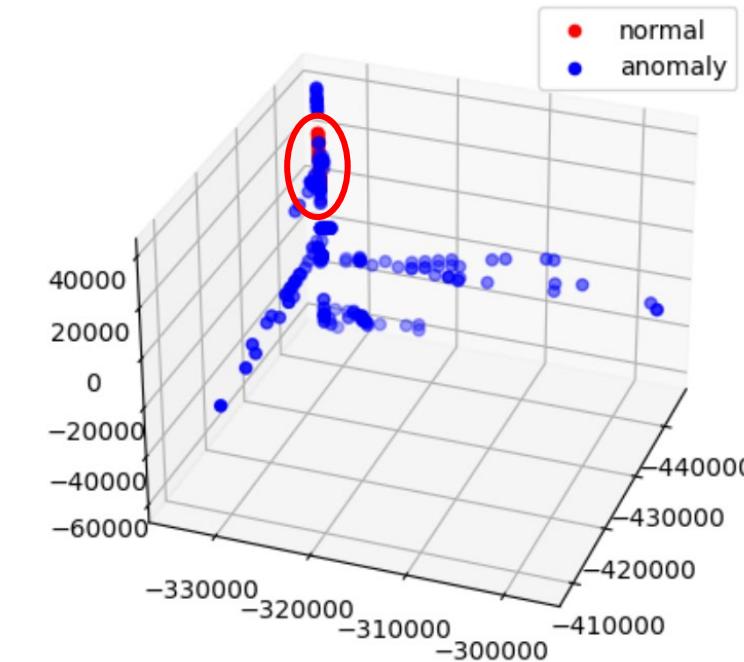
3-dimension으로 변환

- Dataset의 분포를 확인하기 위해서 33차원을 눈으로 확인할 수 있는 2,3차원으로 축소시킬 필요성이 존재, PCA(Eigenvalue Decomposition을 사용한 차원 축소 방법)를 사용하여 Dataset의 차원을 2,3차원으로 축소한 다음 visualize를 진행

# Dataset Visualize 2



2-dimension으로 변환결과



3-dimension으로 변환결과

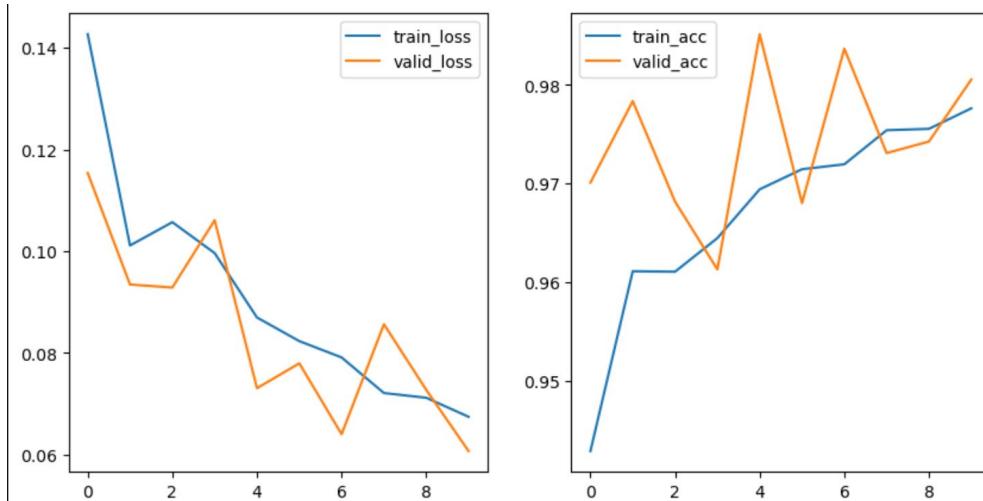
- Anomaly의 경우 5개의 시나리오를 합쳐서 여러 가지의 선으로 표현이 되는 것 처럼 보인다
- 반대로 normal의 경우 간단한 시나리오(웹 서핑, 동영상시청 등)로 구성 되어있기에 분포가 한곳에 몰려 있는 것 처럼 보임

# Model selection case 1: Simple DNN

```
model = nn.Sequential(linblock(32,256),
                      linblock(256,512),
                      linblock(512,1024),
                      linblock(1024,512),
                      linblock(512,256),
                      nn.Linear(256,1),
                      nn.Sigmoid())

model.to(device)
criterion = nn.BCELoss()

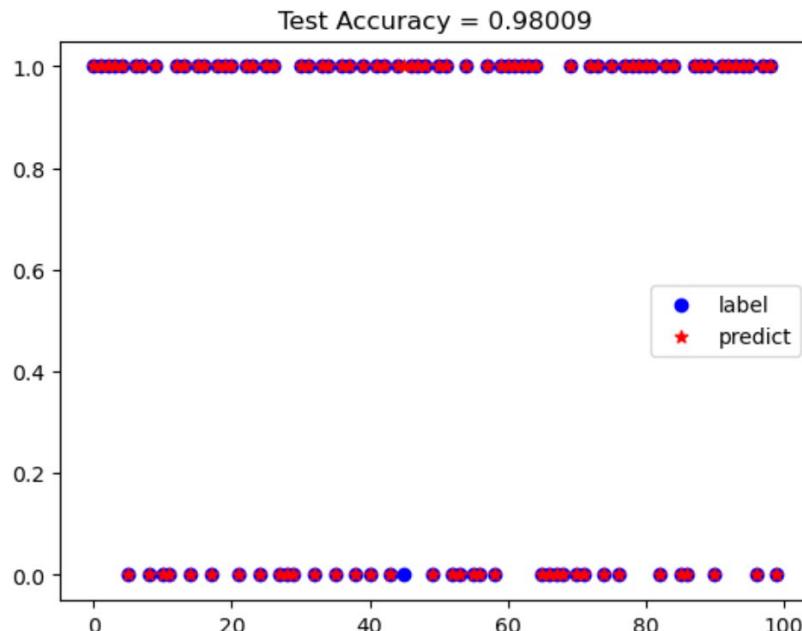
optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.999), eps=1e-08, weight_decay=1e-2)
```



Train Loss: 0.14268 Acc: 0.94291: 100%	397/397 [00:03<00:00, 111.55it/s]
Valid Loss: 0.11540 Acc: 0.97010: 100%	100/100 [00:00<00:00, 162.31it/s]
Train Loss: 0.10114 Acc: 0.96112: 100%	397/397 [00:03<00:00, 111.07it/s]
Valid Loss: 0.09347 Acc: 0.97835: 100%	100/100 [00:00<00:00, 174.50it/s]
Train Loss: 0.10572 Acc: 0.96108: 100%	397/397 [00:03<00:00, 109.82it/s]
Valid Loss: 0.09288 Acc: 0.96817: 100%	100/100 [00:00<00:00, 152.37it/s]
Train Loss: 0.09964 Acc: 0.96448: 100%	397/397 [00:03<00:00, 91.97it/s]
Valid Loss: 0.10612 Acc: 0.96131: 100%	100/100 [00:00<00:00, 176.90it/s]
Train Loss: 0.08699 Acc: 0.96942: 100%	397/397 [00:03<00:00, 109.97it/s]
Valid Loss: 0.07309 Acc: 0.98513: 100%	100/100 [00:00<00:00, 179.80it/s]
Train Loss: 0.08234 Acc: 0.97146: 100%	397/397 [00:03<00:00, 103.95it/s]
Valid Loss: 0.07798 Acc: 0.96802: 100%	100/100 [00:00<00:00, 177.49it/s]
Train Loss: 0.07913 Acc: 0.97195: 100%	397/397 [00:03<00:00, 98.94it/s]
Valid Loss: 0.06405 Acc: 0.98367: 100%	100/100 [00:00<00:00, 179.22it/s]
Train Loss: 0.07214 Acc: 0.97541: 100%	397/397 [00:03<00:00, 110.42it/s]
Valid Loss: 0.08566 Acc: 0.97309: 100%	100/100 [00:00<00:00, 176.41it/s]
Train Loss: 0.07121 Acc: 0.97555: 100%	397/397 [00:03<00:00, 110.03it/s]
Valid Loss: 0.07276 Acc: 0.97426: 100%	100/100 [00:00<00:00, 142.98it/s]
Train Loss: 0.06747 Acc: 0.97762: 100%	397/397 [00:03<00:00, 112.33it/s]
Valid Loss: 0.06073 Acc: 0.98054: 100%	100/100 [00:00<00:00, 160.39it/s]

- 딥러닝 모델의 경우 Time정보와 Source, Destination정보를 drop하였기 때문에 간단한 Fully Connected모델로도 충분하다고 판단하여 위처럼 6개의 hidden layer를 생성
- Loss function은 0 or 1을 구분하는 문제이므로 binary cross entrop를 사용하여 학습을 진행
- Optimizer는 Adam을 사용

# Model selection case 1: Simple DNN



```
from sklearn.model_selection import train_test_split
train,test= train_test_split(dataset, test_size=0.2, random_state=42, stratify=dataset["class"])
train,valid= train_test_split(train, test_size=0.2, random_state=42, stratify=train["class"])

x_test = test.drop("class",axis=1)
y_test = test["class"].copy()
x_train= train.drop('class',axis=1)
y_train= train['class'].copy()

x_train,x_val, y_train,y_val = train_test_split(x_train,y_train, test_size=0.2, random_state=42,stratify=y_train)

#to numpy
x_test = np.array(x_test,dtype=np.float32)
y_test = np.array(y_test,dtype=np.float32)
x_train = np.array(x_train,dtype=np.float32)
x_val = np.array(x_val,dtype=np.float32)
y_train = np.array(y_train,dtype=np.float32)
y_val = np.array(y_val,dtype=np.float32)

train_dataset = customDataset(x_train,y_train)
valid_dataset = customDataset(x_val,y_val)
batch_size = 512
train_loader = DataLoader(train_dataset,batch_size = batch_size,shuffle = True)
valid_loader = DataLoader(valid_dataset,batch_size = batch_size,shuffle = True)
test_dataset = customDataset(x_test,y_test)
test_loader = DataLoader(test_dataset,batch_size = batch_size,shuffle = True)
device = 'cuda'
```

사전에 분리된 Test set에 대해서 예측을 진행

- 학습된 모델을 바탕으로 test 결과 98%의 예측률을 보임

# Model selection case 2: AutoML

```
from pycaret.regression import *
setup_clf = setup(data = train,target='class',train_size=0.7,
                  # numeric_features = ["Duration", "Src Pt", "Dst Pt",],
                  # Remove Perfect Collinearity=True
                  remove_perfect_collinearity=True,
                  session_id=777)
```

Pycaret은 input으로 numpy가 아닌 Dataframe을 넣는다

- 정형 데이터의 경우 AutoML이 성능이 더 좋기에 pycaret이라는 AutoML을 사용하여 모델을 분석

```
#3 가지 좋은 모델을 select
best_model = compare_models(n_select=3)
```

	Model	MAE	MSE	RMSE	R2	RMSLE	MAPE	TT (Sec)
rf	Random Forest Regressor	0.0001	0.0000	0.0046	0.9999	0.0031	0.0001	5.513
et	Extra Trees Regressor	0.0000	0.0000	0.0036	0.9999	0.0025	0.0000	3.692
catboost	CatBoost Regressor	0.0001	0.0000	0.0029	0.9999	0.0019	0.0001	5.825
xgboost	Extreme Gradient Boosting	0.0001	0.0000	0.0035	0.9999	0.0022	0.0001	2.786
dt	Decision Tree Regressor	0.0000	0.0000	0.0053	0.9998	0.0036	0.0000	0.110
lightgbm	Light Gradient Boosting Machine	0.0003	0.0001	0.0076	0.9997	0.0051	0.0002	0.345
gbr	Gradient Boosting Regressor	0.0022	0.0002	0.0135	0.9991	0.0112	0.0011	5.268
knn	K Neighbors Regressor	0.0005	0.0002	0.0154	0.9989	0.0109	0.0004	3.802
ada	AdaBoost Regressor	0.0143	0.0004	0.0195	0.9982	0.0116	0.0189	1.479
br	Bayesian Ridge	0.0084	0.0018	0.0420	0.9918	0.0297	0.0060	0.272
ridge	Ridge Regression	0.0085	0.0018	0.0421	0.9918	0.0297	0.0061	0.036
omp	Orthogonal Matching Pursuit	0.1259	0.0414	0.2035	0.8078	0.1586	0.0604	0.039
lr	Linear Regression	0.2029	0.0708	0.2660	0.6716	0.1965	0.1478	0.425
lasso	Lasso Regression	0.2527	0.1022	0.3196	0.5258	0.2239	0.2055	1.254
en	Elastic Net	0.2524	0.1022	0.3196	0.5258	0.2236	0.2052	1.495
huber	Huber Regressor	0.2697	0.1131	0.3362	0.4753	0.2260	0.2467	2.194
llar	Lasso Least Angle Regression	0.4309	0.2155	0.4642	-0.0000	0.3251	0.3142	0.027
dummy	Dummy Regressor	0.4309	0.2155	0.4642	-0.0000	0.3251	0.3142	0.026
lar	Least Angle Regression	0.0150	2.0137	0.5452	-8.3639	0.0379	0.0135	0.043
par	Passive Aggressive Regressor	3.2667	132.9917	8.3033	-616.3274	0.9000	2.2321	0.139

Pycaret은 input으로 numpy가 아닌 Dataframe을 넣는다

# Model selection case 2: AutoML

```
# 전체 데이터로 학습한 모델을 기반으로 test에 대해서 예측을 진행  
predictions = predict_model(final_model, data = test)
```

predictions																					
	Length	Sequence Number	Acknowledgment Number	Window	Type	Time to Live	Total Length	Code	Length.1	Source_Port	...	0x012	0x014	0x018	0x019	0x052	0x0c2	RST_TTL	RST_Total_length	class	Label
130422	54	1.0	1.0	0.0	-1.0	64	40	-1.0	-1.0	44420.0	...	0	1	0	0	0	0	-1	-1	1	1.000003e+00
50035	60	0.0	0.0	1024.0	-1.0	51	44	-1.0	-1.0	40609.0	...	0	0	0	0	0	0	-1	-1	1	1.000002e+00
75857	60	0.0	0.0	1024.0	-1.0	51	44	-1.0	-1.0	40609.0	...	0	0	0	0	0	0	-1	-1	1	9.999909e-01
164887	78	0.0	0.0	65535.0	-1.0	254	64	-1.0	-1.0	50515.0	...	0	0	0	0	0	0	-1	-1	1	1.000053e+00
230818	54	1.0	1.0	0.0	-1.0	64	40	-1.0	-1.0	31654.0	...	0	1	0	0	0	0	-1	-1	1	1.000002e+00
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
42130	54	1.0	1.0	0.0	-1.0	64	40	-1.0	-1.0	45920.0	...	0	1	0	0	0	0	-1	-1	1	9.999985e-01
124743	60	0.0	0.0	1024.0	-1.0	37	44	-1.0	-1.0	40609.0	...	0	0	0	0	0	0	-1	-1	1	1.000104e+00
172180	54	1.0	1.0	0.0	-1.0	64	40	-1.0	-1.0	65449.0	...	0	1	0	0	0	0	-1	-1	1	1.000008e+00
394756	74	-1.0	-1.0	-1.0	-1.0	64.0	60.0	-1.0	40.0	32923.0	...	0	0	0	0	0	0	-1	-1	0	8.860698e-06
390675	1399	-1.0	-1.0	-1.0	-1.0	255.0	1385.0	-1.0	1365.0	443.0	...	0	0	0	0	0	0	-1	-1	0	1.349500e-07

- AutoML로 찾아낸 모델에 대해서 ensemble을 진행하고 Test set에 대해서 예측을 진행하니 Label이라는 column이 새로 생성됨

# Model selection case 2: AutoML

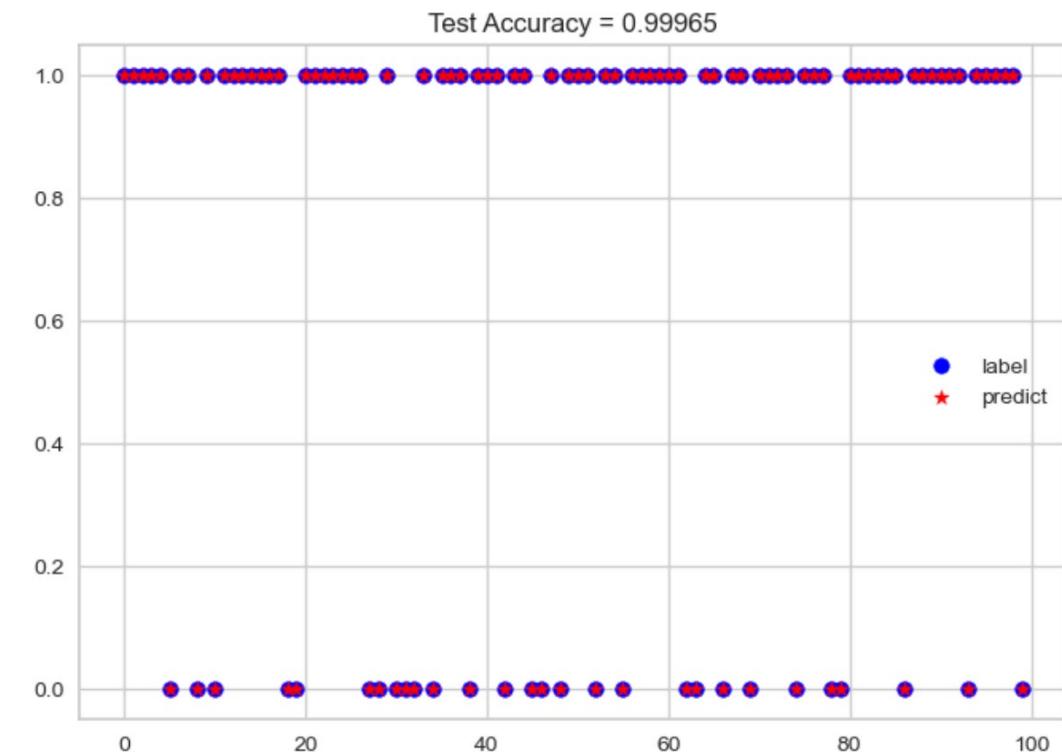
```
automl_predict = np.array(predictions["Label"])
automl_predict
```

```
array([1.0000293e+00, 1.0000233e+00, 9.99990939e-01, ...,
       1.00000751e+00, 8.86069839e-06, 1.34949980e-07])
```

```
#output이 float이므로 정수형으로 형변환
automl_predict = np.round(automl_predict,2)
automl_predict
```

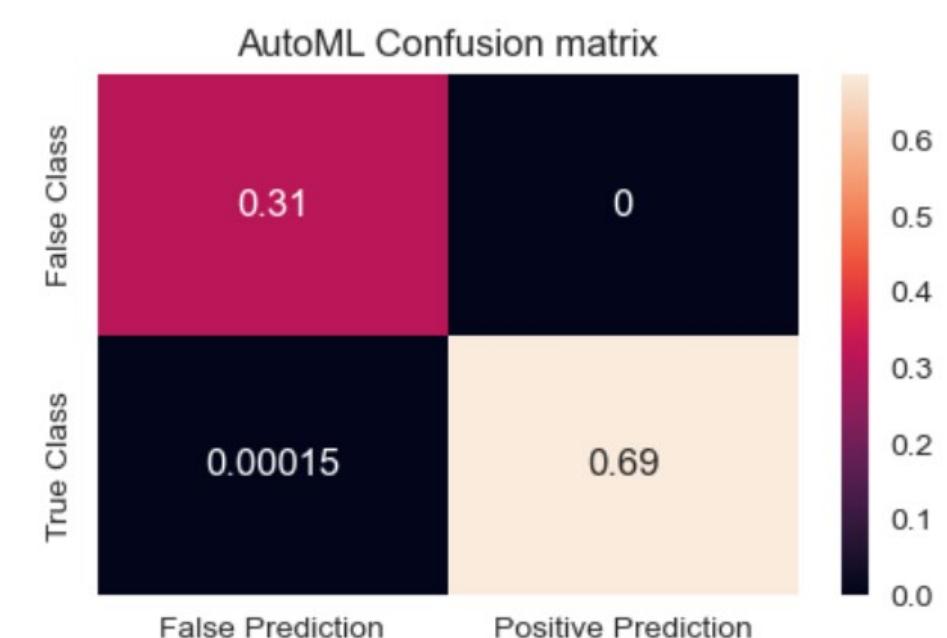
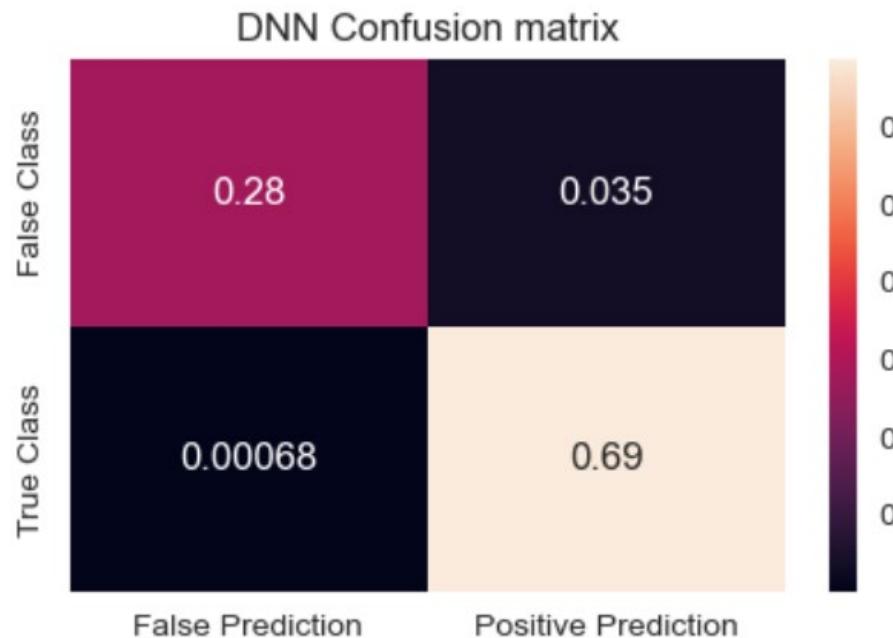
```
array([1., 1., 1., ..., 1., 0., 0.])
```

Label에 맞게 형변환을 진행하고 원래 label과 비교해보면



학습된 모델을 바탕으로 test 결과 99%의 예측률을 보임

# Additional Metric



추가적으로 Confusion matrix 와 Precision, Recall score를 출력하면 다음과 같았다

DNN Precision score	0.95125
AutoML Precision score	1.00000
DNN Recall score	0.99901
AutoML Recall score	0.99978

# Conclusion

---

1. 주어진 Dataset을 사용한 것이 아닌 wireshark를 사용하여 직접 수집한 패킷에 대해서 anomaly detection을 진행하는 딥러닝 모델 개발
2. Input으로 들어가는 Column의 feature에 맞게 패킷을 수집하면 기존의 특별한 규칙으로 진행되는 anomaly detection이 아닌 딥러닝 기반의 Anomaly Detection을 진행
3. 5가지 network attack 시나리오에 대해서 학습을 진행하였고, 추가적인 시나리오가 있다면 추가적으로 학습이 가능한 시스템 설계
4. Wireshark로 수집한 데이터에 대한 preprocessing pipeline제안
5. 직접 설계한 DNN뿐만 아니라 AutoML로 최적화된 모델을 찾아서 Anomaly Detection을 진행

# Limitation

---

1. 같은 환경에서 test한 시나리오가 아니기 때문에 Time정보를 사용할 수 없었음 , time정보를 사용할 수 있다면 LSTM, GRU등 RNN기법의 딥러닝 모델도 사용할 수 있었을 텐데 시간 정보를 사용할 수 없었던 것이 아쉬움
2. Dataset의 Source, Destination정보를 사용할 수 없었음, 예를 들어 fin\_scan같은 경우 Source(공격자)가 10.40.219.42로 사설 IP임에도 해당 ip로 들어오는 정보를 모두 Anomaly라고 탐지할 수 있는 위험성이 존재  
Source, Destination의 IPv4정보를 버릴 수 밖에 없었음 IP정보를 잘 처리할 수 있는 기법이 있었다면 더 좋은 탐지 기법(Graph Neural Network)등으로 표현할 수 있었을 텐데 아쉬움이 있음
3. AutoML을 처음 사용해 보았는데, DNN의 경우 Binary classification문제라 loss function으로 BCE를 쓰는 등 직접 문제를 설정하여 해결하는 등 분석을 할 수 있었지만, AutoML의 경우 좋은 모델을 알아서 찾고, metric역시 알아서 해주기에 이게 왜 좋은지를 설명하기 어려움이 있었음