Routing network traffic based on firewall logs using Machine Learning

1. Overview:

1.1. Introduction:

The Internet is such an essential part of our lives nowadays, that it can be considered as one of the necessities to live a better life. However, with that said, it is also necessary that it is not being used for illegal activities and more so in private networks such as university campuses or corporate offices to protect and secure confidential data. That's where the firewall comes into picture. A firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predefined security rules. Hence it can be thought of as a safeguard that establishes a barrier between a trusted network (i.e. campus or office) and an untrusted network (i.e. internet).

1.2. Business Problem:

The important part of firewall systems is obviously the security rules governing the network traffic. The proper routing of the traffic ensures that the network is secured with security policy and users don't experience any unwanted hassle to access allowed web pages. Setting up a firewall system is a complex and an error-prone task as it is carried out by the administrators of the network and hence, they may be subject to human error and moreover with time they may be required to be modified to remain compliant with the security policy.

For instance, if a person wants to access a prohibited website on the network under consideration, the firewall should be able to block that particular website on the network. Corporate or university networks for that

matter might have strict reservations about the fraudulent websites that should not be accessed by its employees or students. Given the vast majority of the internet, it is really a cumbersome task to maintain these rules for a team of firewall administrators as any mistake by them may incur significant penalty or even exposure of confidential data to the outside network. To eliminate these errors and to provide stable internet traffic, the proper maintenance of the firewall becomes very important with regards to these networks. As the historical data of network traffic can be easily obtained from firewall log reports, machine learning models can be of great help to solve this problem.

1.3. ML Problem Formulation:

The mentioned case study focuses on enabling the firewall system to learn and implement these security rules dynamically based on historically logged data. As the log data is easily available from the firewall systems, machine learning models can be implemented effectively for the aforementioned problem. The dataset for this case study is provided by Fatih Ertam, Firat University, Turkey at UCI machine learning repository. The case study will design a model to predict whether the underlying traffic should be allowed based on log data for the particular network communication using various machine learning techniques.

1.4. Business Constraints:

The machine learning model designed should be able to quickly predict the action to be taken in order to avoid an unwanted hindrance to the network latency as it might be counterproductive to the network performance. In short, irrespective of the training time complexity the test time complexity must be low. Also, the network administrators might want some form of explainability for the predicted action classes by the model,

hence the model should also be able to provide importance of features in predictions.

1.5. Dataset:

The dataset is compiled by Fatih Ertam at Firat University, Turkey and can be downloaded from the link given below-

- Internet Firewall dataset

The dataset contains 65532 instances with 12 features collected by logs of the university firewall system. The summary of the features is given below-

Sr. No.	Feature Name	Description
1.	Source Port	Client Source Port
2.	Destination Port	Client Destination Port
3.	NAT Source Port	Network Address Translation Source Port
4.	NAT Destination Port	Network Address Translation Destination Port
5.	Elapsed Time (sec)	Elapsed Time for flow
6.	Bytes	Total Bytes
7.	Bytes Sent	Bytes Sent
8.	Bytes Received	Bytes Received
9.	Packets	Total Packets
10.	pkts_sent	Packets Sent
11.	pkts_received	Packets Received
12.	Action	Action to be taken on traffic

The action which is our response variable is categorized in four classes.

Namely-

Sr. No.	Action	Description
1.	Allow	Allows the internet traffic.
2.	Deny	Blocks traffic and enforces the default Deny Action defined for the application that is being denied.
3.	Drop	Silently drops the traffic; for an application, it overrides the default deny action. A TCP reset is not sent to the host/application.
4.	Reset-Both	Sends a TCP reset to both the client-side and server-side devices

2. Exploratory Data Analysis (EDA):

We will carry out exploratory data analysis to investigate and visualize the dataset using visualization libraries such as Matplotlib and Seaborn. This will help us gain important insights on the relationship between features and response variable. Let us now import the required libraries

2.1. Importing required libraries:

In [3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
title_font = {'family': 'serif', 'color': 'darkred', 'weight': 'bold', 'size': 18}
label_font = {'family': 'Arial', 'weight': 'normal', 'size': 16}
import warnings
warnings.filterwarnings("ignore")
```

2.2. Reading the dataset:

In [4]:
```
data = pd.read_csv("log2.csv")
data.head()
```

Out[4]:

	Source Port	Destination Port	NAT Source Port	NAT Destination Port	Action	Bytes	Bytes Sent	Bytes Received	Packets	Elapsed Time (sec)	pkts_se
0	57222	53	54587	53	allow	177	94	83	2	30	
1	56258	3389	56258	3389	allow	4768	1600	3168	19	17	
2	6881	50321	43265	50321	allow	238	118	120	2	1199	
3	50553	3389	50553	3389	allow	3327	1438	1889	15	17	
4	50002	443	45848	443	allow	25358	6778	18580	31	16	

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65532 entries, 0 to 65531
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Source Port           65532 non-null  int64
 1   Destination Port      65532 non-null  int64
 2   NAT Source Port       65532 non-null  int64
 3   NAT Destination Port  65532 non-null  int64
 4   Action                65532 non-null  object
 5   Bytes                 65532 non-null  int64
 6   Bytes Sent            65532 non-null  int64
 7   Bytes Received        65532 non-null  int64
 8   Packets               65532 non-null  int64
 9   Elapsed Time (sec)    65532 non-null  int64
 10  pkts_sent             65532 non-null  int64
 11  pkts_received         65532 non-null  int64
dtypes: int64(11), object(1)
memory usage: 6.0+ MB
```

The dataset does not have any missing values. Also, Ports cannot be considered as numerical features. Hence, they should be converted into object data type.

In [6]:
```
port_features = ['Source Port', 'Destination Port', 'NAT Source Port', 'NAT Destination
for port in port_features:
    data[port] = data[port].astype('object')
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65532 entries, 0 to 65531
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Source Port           65532 non-null  object
```

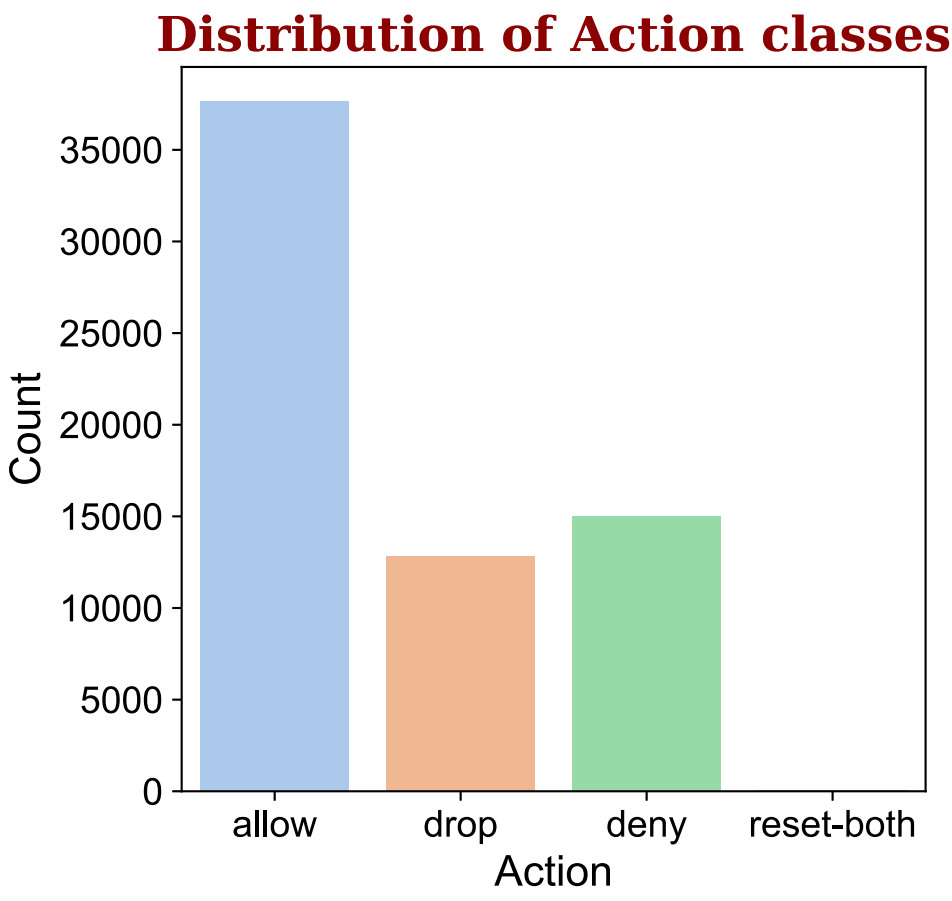
```
 1   Destination Port      65532 non-null  object
 2   NAT Source Port       65532 non-null  object
 3   NAT Destination Port  65532 non-null  object
 4   Action                65532 non-null  object
 5   Bytes                 65532 non-null  int64
 6   Bytes Sent            65532 non-null  int64
 7   Bytes Received        65532 non-null  int64
 8   Packets               65532 non-null  int64
 9   Elapsed Time (sec)    65532 non-null  int64
 10  pkts_sent             65532 non-null  int64
 11  pkts_received         65532 non-null  int64
dtypes: int64(7), object(5)
memory usage: 6.0+ MB
```

2.3. Univariate Analysis:

2.3.1. Analysis of response variable **`Action`**.

In [7]:

```
fig = plt.figure(figsize = (5, 5))
ax = sns.countplot(x = "Action", data = data, palette = 'pastel')
plt.title("Distribution of Action classes", fontdict = title_font)
plt.xlabel("Action", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
# Set the tick labels font referred from (https://stackoverflow.com/a/23572192)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

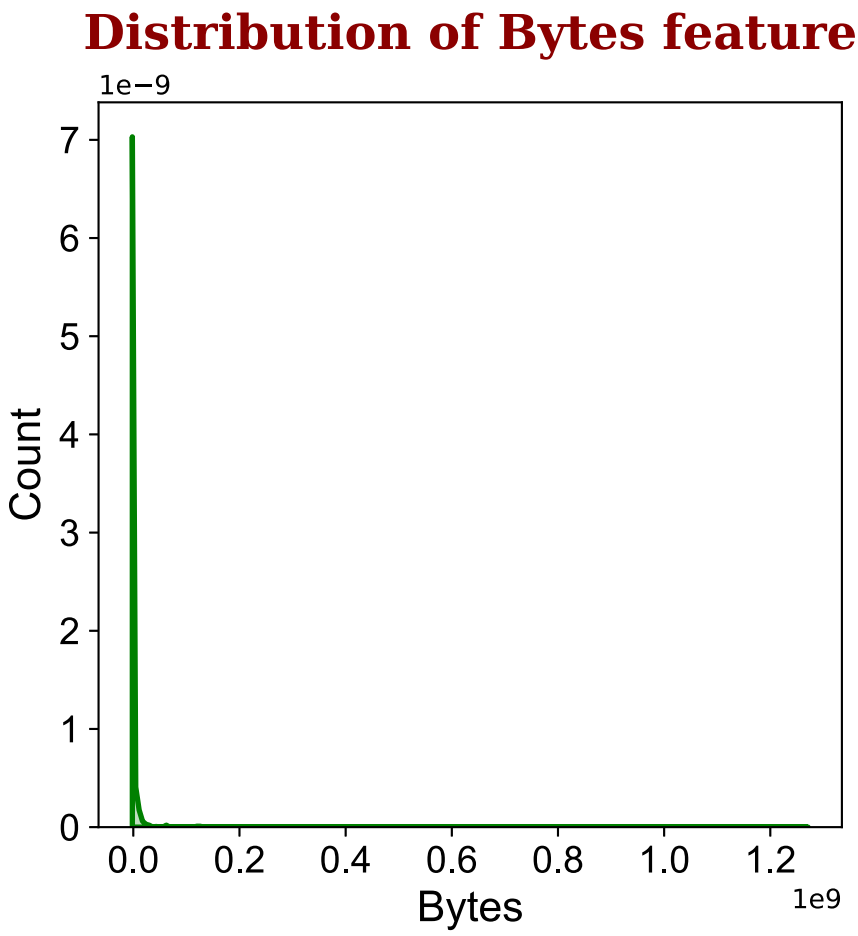


The plot shows that the dataset contains four classes of response

variable with imbalanced distribution among them. Action 'allow' is the most frequent while, 'reset-both' is rare. This is to be expected as traffic will be allowed most of the time and once in while 'reset-both' which resets client-server networks will be required.

2.3.2. Analysis of `Bytes` feature.

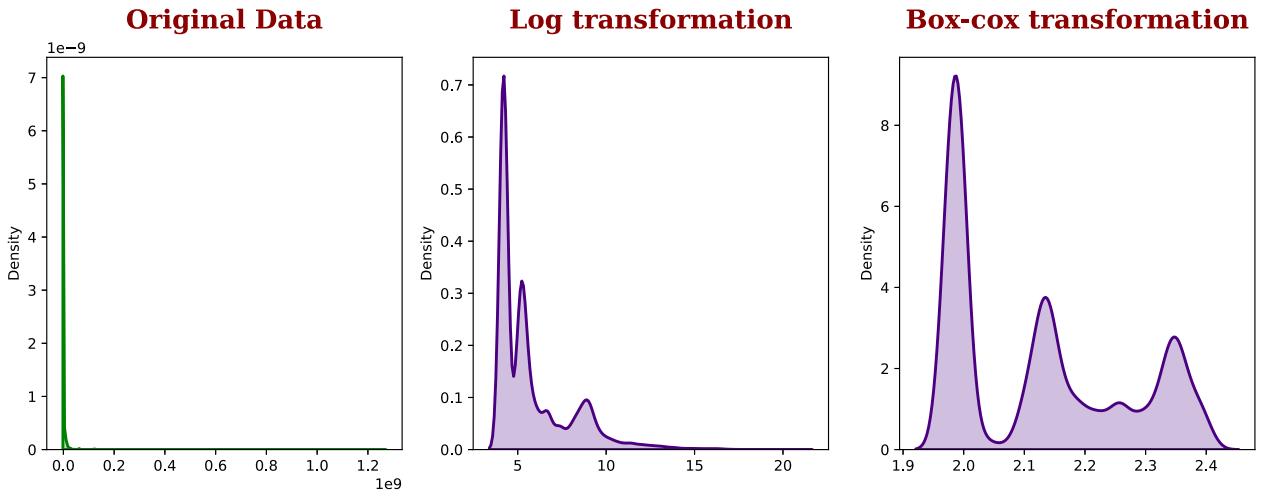
In [8]:
```
plt.figure(figsize = (5, 5))
ax = sns.distplot(data['Bytes'].values, hist = False, kde = True, kde_kws = {'shade': T
plt.title("Distribution of Bytes feature", fontdict = title_font, pad = 20.0)
plt.xlabel("Bytes", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```



The plot shows that the `Bytes` feature is extremely right-skewed. This may be because of the presence of outliers. First, let us check whether it can be transformed into Gaussian distribution using log transformation or Box-cox transformation.

In [9]:

```
from scipy import stats
original_data = data['Bytes'].values

# applying log tranbsformation
log_data = np.log(original_data)
# applying Box-cox transformation
box_cox_data, lambda_ = stats.boxcox(original_data)

# creating axes to draw plots
fig, ax = plt.subplots(1, 3)

# plotting the distributions
sns.distplot(original_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewi
sns.distplot(log_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewidth':
sns.distplot(box_cox_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewid

# adding titles to the subplots
ax[0].set_title("Original Data", fontdict = title_font, pad = 20.0)
ax[1].set_title("Log transformation", fontdict = title_font, pad = 20.0)
ax[2].set_title("Box-cox transformation", fontdict = title_font, pad = 20.0)

# rescaling the figure
fig.set_figheight(5)
fig.set_figwidth(15)
```



The plot shows that the **`Bytes`** feature has multimodal distribution and hence it cannot be transformed into normal diistribution. It also has outliers and hence, needs to be investigated further. The 90th to 100th percentile values can give us a better idea about the outliers.

In [10]:

```
for i in range(90, 101):
    print(f"{i}th percentile value of Bytes feature is {int(np.percentile(data['Bytes']
```

```
90th percentile value of Bytes feature is 8007
91th percentile value of Bytes feature is 8820
92th percentile value of Bytes feature is 9841
93th percentile value of Bytes feature is 11485
94th percentile value of Bytes feature is 14558
```

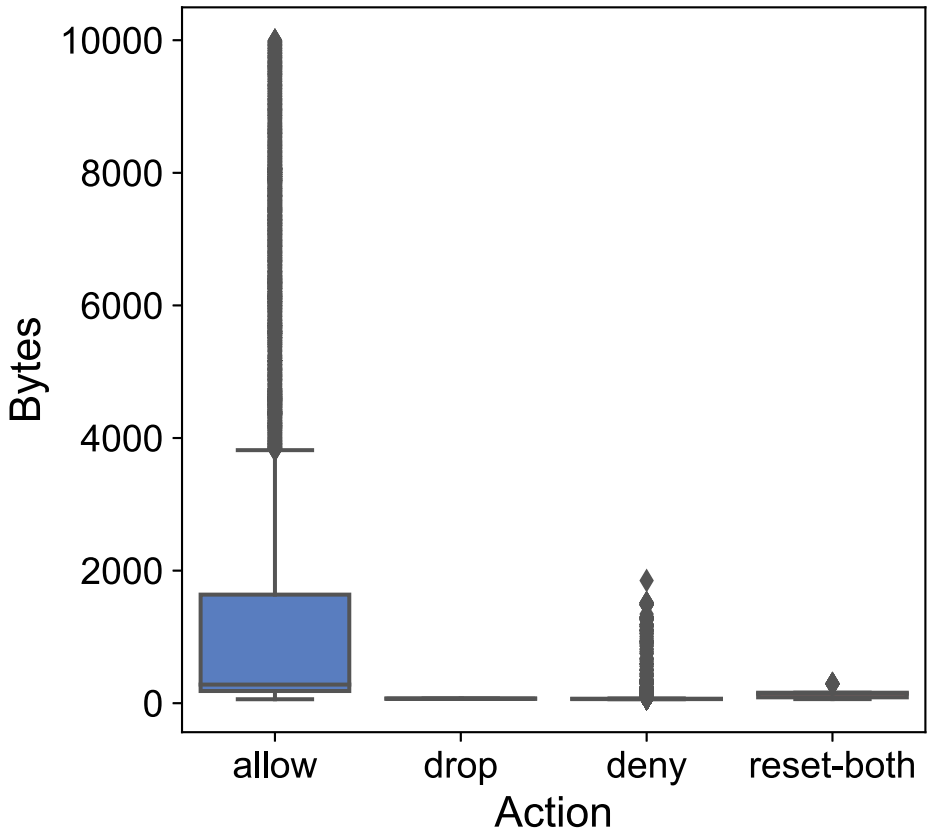
```
95th percentile value of Bytes feature is 20411
96th percentile value of Bytes feature is 32218
97th percentile value of Bytes feature is 64387
98th percentile value of Bytes feature is 152192
99th percentile value of Bytes feature is 520001
100th percentile value of Bytes feature is 1269359015
```

We can observe that more than 92 percent instances have **`Bytes`** less than 10000 and 100th percentile value is causing huge skew in the distribution of **`Bytes`** feature. Also, we can ignore the **`Bytes`** values beyond 99th percentile. Nonetheless, the **`Bytes`** feature is extremely right skewed. Let us check the relationship of **`Bytes`** feature with response variable **`Action`**. For the sake of interpretability, We will consider values below 10000, as it covers almost 93 percent dataset.

In [11]:
```
# removing instances where Bytes are beyond 99th percentile
data = data[data['Bytes'] <= np.percentile(data['Bytes'], 99)]
```

In [12]:
```
plt.figure(figsize = (5, 5))
ax = sns.boxplot(x = 'Action', y = 'Bytes', data = data[data['Bytes'] < 10000], palette
plt.title("Bytes feature vs Action class", fontdict = title_font, pad = 20.0)
plt.xlabel("Action", fontdict = label_font)
plt.ylabel("Bytes", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

Bytes feature vs Action class



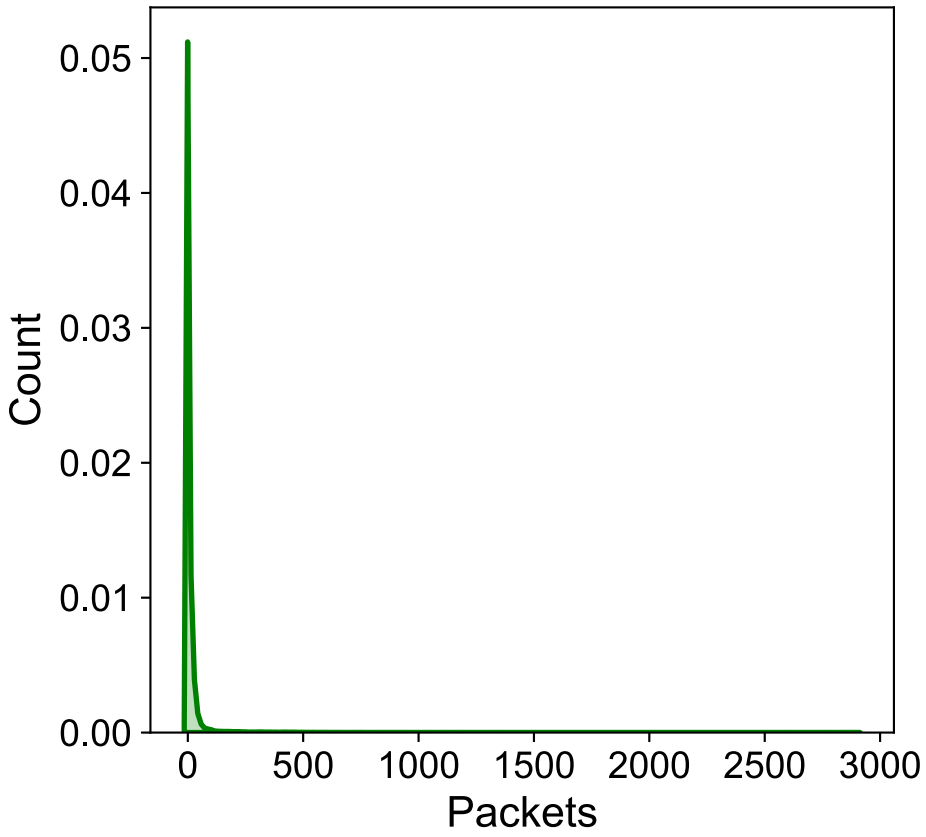
We can observe that higher the number of **`Bytes`** more likely it is that the traffic will be allowed. But nothing can be said about other three **`Action`** classes based on just **`Bytes`** feature.

2.3.3. Analysis of **`Packets`** feature.

In [13]:

```
plt.figure(figsize = (5, 5))
ax = sns.distplot(data['Packets'].values, hist = False, kde = True, kde_kws = {'shade':
plt.title("Distribution of Packets feature", fontdict = title_font, pad = 20.0)
plt.xlabel("Packets", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

Distribution of Packets feature

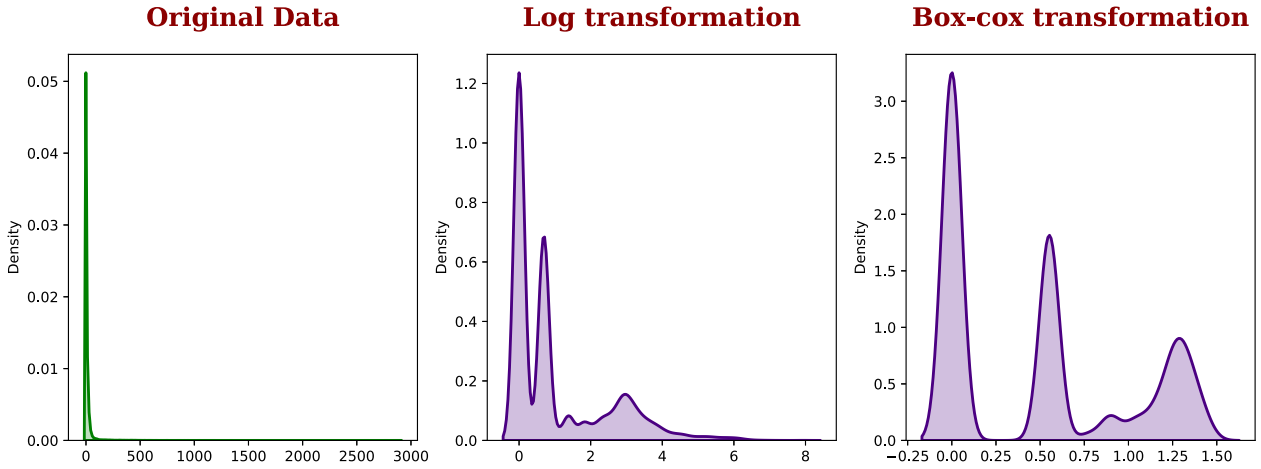


The plot shows that the `Packets` feature is extremely right-skewed. This may be because of the presence of outliers. First, let us check whether it can be transformed into Gaussian distribution using log transformation or Box-cox transformation.

In [14]:

```
from scipy import stats
original_data = data['Packets'].values

# applying log tranbsformation
log_data = np.log(original_data)
# applying Box-cox transformation
box_cox_data, lambda_ = stats.boxcox(original_data)

# creating axes to draw plots
fig, ax = plt.subplots(1, 3)

# plotting the distributions
sns.distplot(original_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewi
sns.distplot(log_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewidth':
sns.distplot(box_cox_data, hist = False, kde = True, kde_kws = {'shade': True, 'linewid

# adding titles to the subplots
ax[0].set_title("Original Data", fontdict = title_font, pad = 20.0)
ax[1].set_title("Log transformation", fontdict = title_font, pad = 20.0)
ax[2].set_title("Box-cox transformation", fontdict = title_font, pad = 20.0)
```

```
# rescaling the figure
fig.set_figheight(5)
fig.set_figwidth(15)
```



The plot shows that the **`Packets`** feature has multimodal distribution and hence it cannot be transformed into Gaussian distribution. It also has outliers and hence, needs to be investigated further. The 90th to 100th percentile values can give us a better idea about the outliers.

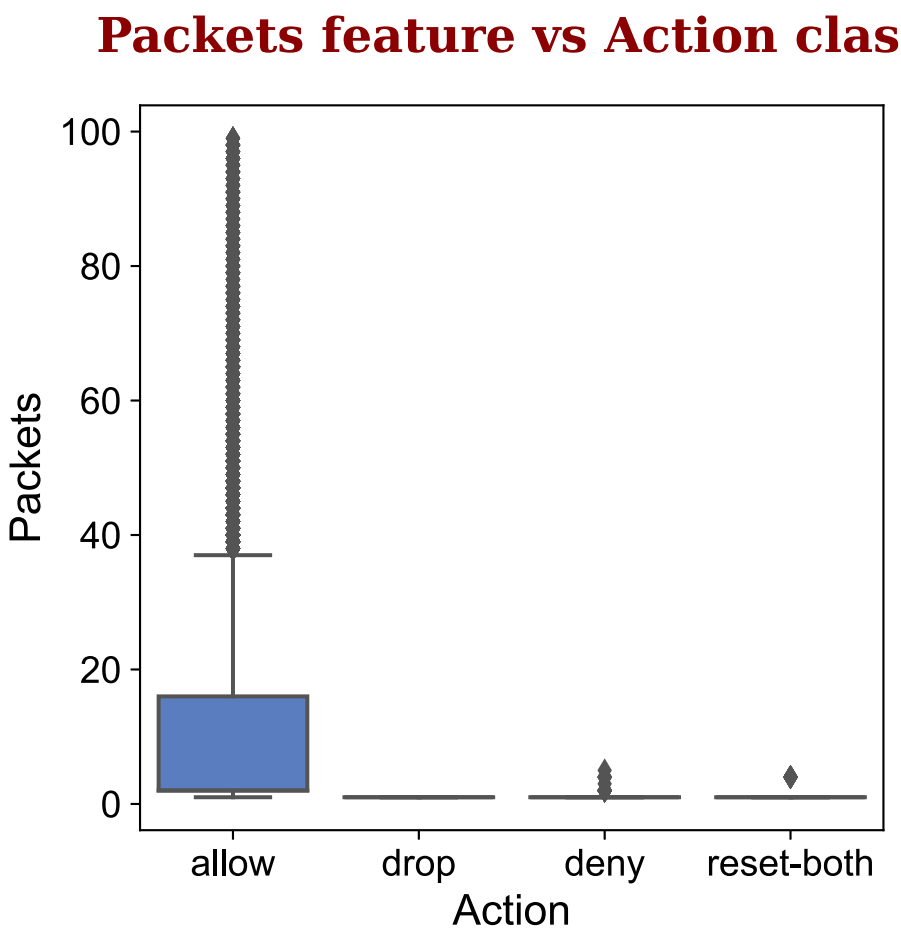
In [15]:
```
for i in range(90, 101):
    print(f"{i}th percentile value of Packets feature is {int(np.percentile(data['Packe
```

```
90th percentile value of Packets feature is 24
91th percentile value of Packets feature is 26
92th percentile value of Packets feature is 28
93th percentile value of Packets feature is 31
94th percentile value of Packets feature is 36
95th percentile value of Packets feature is 42
96th percentile value of Packets feature is 50
97th percentile value of Packets feature is 66
98th percentile value of Packets feature is 103
99th percentile value of Packets feature is 211
100th percentile value of Packets feature is 2897
```

We can observe that more than 97 percent instances have **`Packets`** less than 100 and 100th percentile value is causing huge skew in the distribution of **`Packets`** feature. Also, we can ignore the **`Packets`** values beyond 99th percentile. Nonetheless, the **`Packets`** feature is extremely right skewed. Let us check the relationship of **`Packets`** feature with response variable **`Action`**. For the sake of interpretability, We will consider values below 100, as it covers almost 98 percent dataset.

In [16]:
```
# removing instances where Packets are beyond 99th percentile
data = data[data['Packets'] <= np.percentile(data['Packets'], 99)]
```

In [17]:
```
plt.figure(figsize = (5, 5))
ax = sns.boxplot(x = 'Action', y = 'Packets', data = data[data['Packets'] < 100], palet
plt.title("Packets feature vs Action class", fontdict = title_font, pad = 20.0)
plt.xlabel("Action", fontdict = label_font)
plt.ylabel("Packets", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```



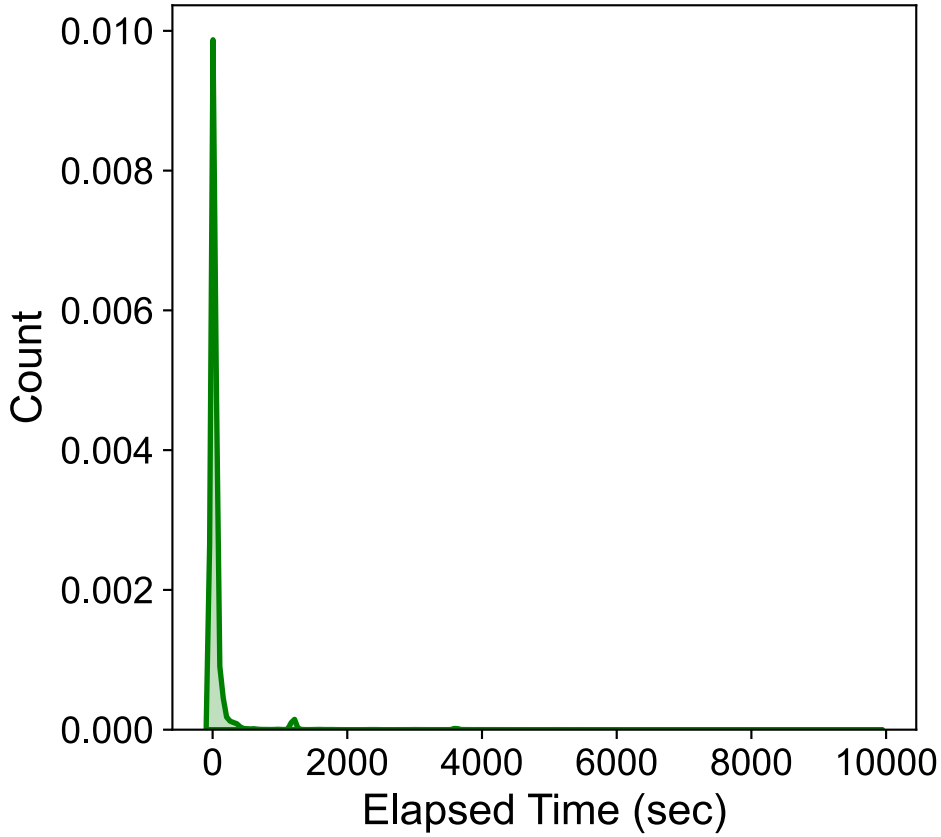
We can observe that higher the number of **`Packets`** more likely it is that the traffic will be allowed. But nothing can be said about other three **`Action`** classes based on just **`Packets`** feature.

2.3.4. Analysis of **`Elapsed Time (sec)`** feature.

In [18]:
```
plt.figure(figsize = (5, 5))
ax = sns.distplot(data['Elapsed Time (sec)'].values, hist = False, kde = True, kde_kws
plt.title("Distribution of Elapsed Time (sec) feature", fontdict = title_font, pad = 20
plt.xlabel("Elapsed Time (sec)", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
```

```
    label.set_fontsize(14)
plt.show()
```

Distribution of Elapsed Time (sec) feature



The plot shows that the **`Elapsed Time (sec)`** feature is extremely right-skewed and has multimodal distribution. Hence, it cannot be transformed into Gaussian distribution. It also has outliers and hence, needs to be investigated further. The 90th to 100th percentile values can give us a better idea about the outliers.

In [19]:
```
for i in range(90, 101):
    print(f"{i}th percentile value of Elapsed Time (sec) feature is {int(np.percentile(
```

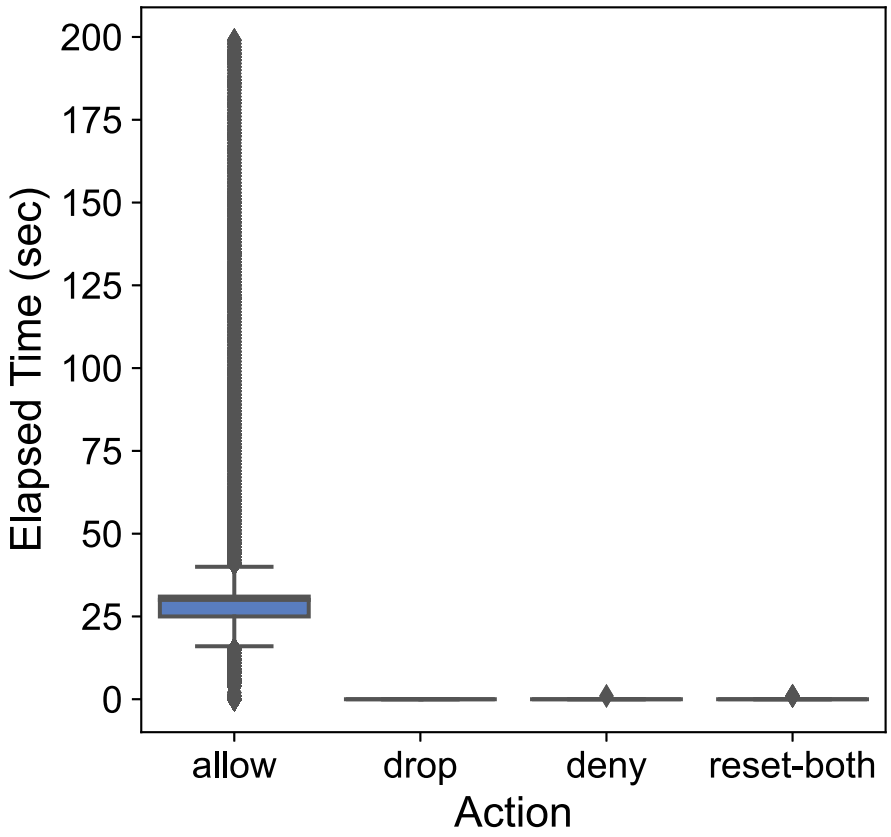
```
90th percentile value of Elapsed Time (sec) feature is 89
91th percentile value of Elapsed Time (sec) feature is 118
92th percentile value of Elapsed Time (sec) feature is 121
93th percentile value of Elapsed Time (sec) feature is 134
94th percentile value of Elapsed Time (sec) feature is 143
95th percentile value of Elapsed Time (sec) feature is 179
96th percentile value of Elapsed Time (sec) feature is 251
97th percentile value of Elapsed Time (sec) feature is 324
98th percentile value of Elapsed Time (sec) feature is 710
99th percentile value of Elapsed Time (sec) feature is 1200
100th percentile value of Elapsed Time (sec) feature is 9851
```

We can observe that more than 95 percent instances have **`Elapsed Time (sec)`** less than 200 and 100th percentile value is causing huge skew in the distribution of **`Elapsed Time (sec)`** feature. Also, we can ignore values beyond 99th percentile. Nonetheless, the **`Elapsed Time (sec)`** feature is extremely right skewed. Let us check the relationship of **`Elapsed Time (sec)`** feature with response variable **`Action`**. For the sake of interpretability, We will consider values below 200, as it covers almost 96 percent dataset.

In [20]:
```
# removing instances where Elapsed Time (sec) are beyond 99th percentile
data = data[data['Elapsed Time (sec)'] <= np.percentile(data['Elapsed Time (sec)'], 99)
```

In [21]:
```
plt.figure(figsize = (5, 5))
ax = sns.boxplot(x = 'Action', y = 'Elapsed Time (sec)', data = data[data['Elapsed Time
plt.title("Elapsed Time (sec) vs Action class", fontdict = title_font, pad = 20.0)
plt.xlabel("Action", fontdict = label_font)
plt.ylabel("Elapsed Time (sec)", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

Elapsed Time (sec) vs Action class



We can observe that higher the number of `Elapsed Time (sec)` more likely it is that the traffic will be allowed. But nothing can be said about other three `Action` classes based on just `Elapsed Time (sec)` feature.

2.4. Bivariate Analysis:

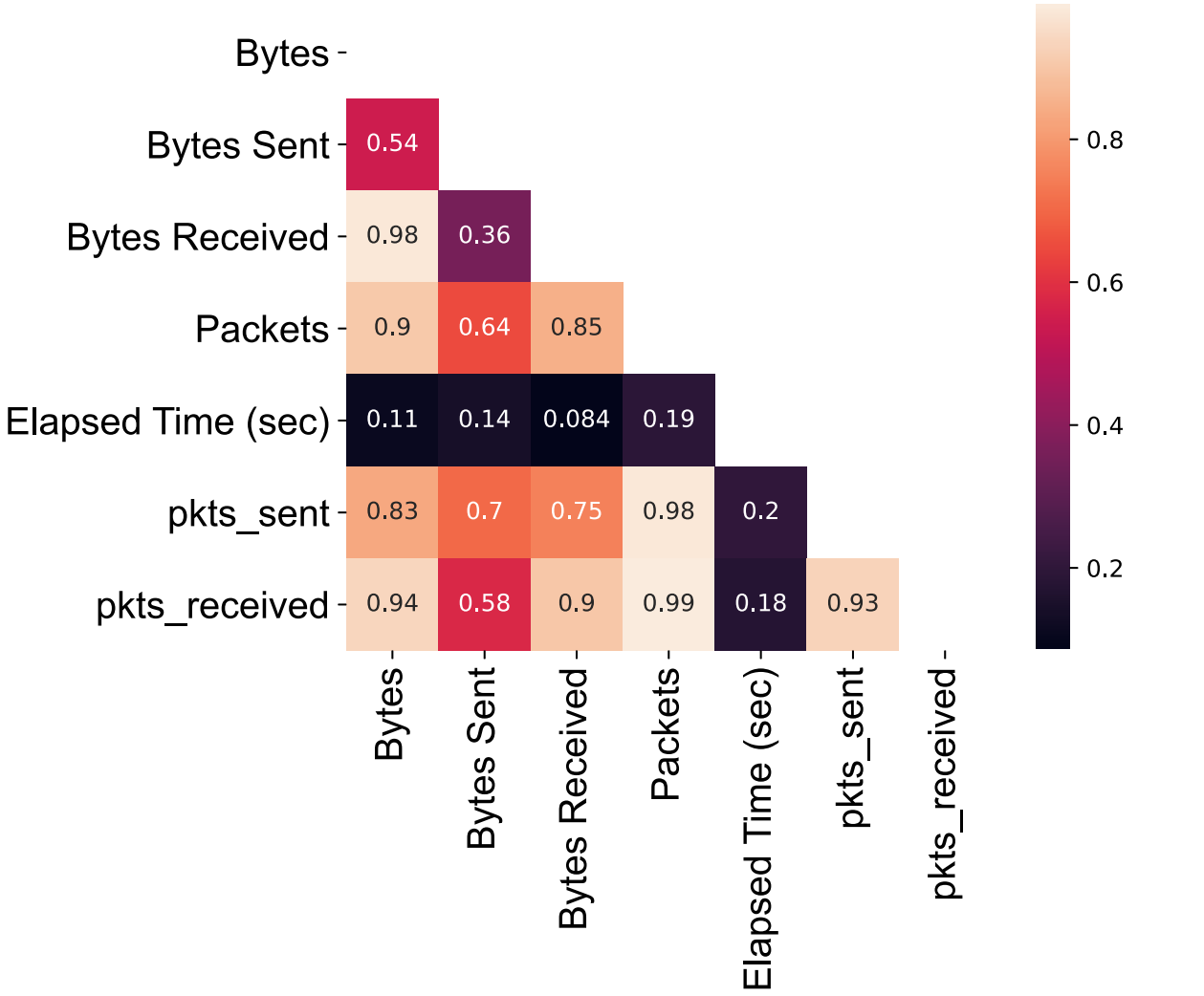
2.4.1. correlation heatmap

In bivariate analysis, we will check for relationships between pairs of variables. The correlation is a good measure for this purpose. Let us plot the correlation heatmap between all the numerical features.

In [22]:

```
# selecting numerical features from the dataset
numerical = data.select_dtypes(include=['int64'])

# calculating correlation matrix
plt.figure(figsize=(7, 5))
correlation = numerical.corr()

# applying mask to crop the heatmap
```

```
mask = np.zeros_like(correlation)
mask[np.triu_indices_from(mask)] = True
ax = sns.heatmap(correlation, mask=mask, annot = True, square=True)

# formatting labels and title
ax.set_xticklabels(ax.get_xmajorticklabels(), fontdict = label_font, rotation = 90)
ax.set_yticklabels(ax.get_ymajorticklabels(), fontdict = label_font)
plt.title("Correlation heatmap of numerical features", fontdict = title_font, pad = 20.
plt.show()
```



The plot shows correlation between the pairs of numerical features in the dataset. As we might expect, features **`Packets`**, **`pkts_sent`** and **`pkts__received`** are highly correlated with each other. Similarly, **`Bytes`** and **`Bytes Received`** are also highly correlated with each other. But, that is not the case with **`Bytes`** and **`Bytes Sent`**. This may be due to inconsistent network connectivity. Also, **`Elapsed Time (sec)`** is not correlated with other fetures, which is obvious as it is not a data

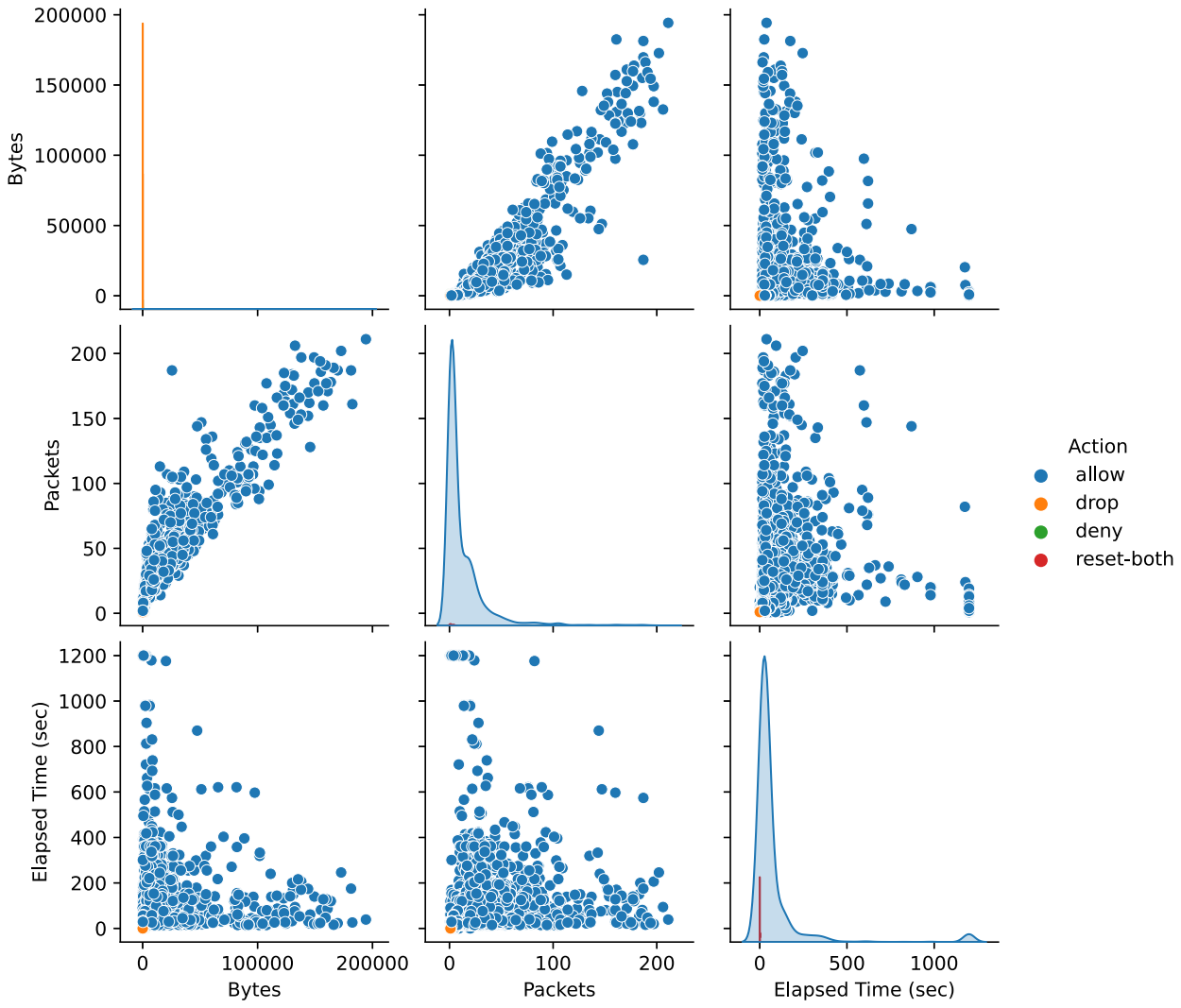
communication unit like **Bytes** or **Packets** which are also highly correlated with each other.

2.4.2. Pairplots between **Bytes**, **Packets**, **Elapsed Time (sec)** features

Now, we can check for the dependence of **Action** class on the pair of variables by using scatterplots between the pairs of variables.

In [23]:
```
plt.figure(figsize = (10, 10))

# stratified sampling of 10000 points based on Action column referred from: (https://ww
sample_df = data[['Bytes', 'Packets', 'Elapsed Time (sec)', 'Action']]
N = 10000
sample_df = sample_df.groupby('Action', group_keys=False).apply(lambda x: x.sample(int(

# plotting pairplot using random sample of 10000 datapoints to limit the file size
ax = sns.pairplot(data = sample_df, hue = 'Action')
plt.show()
```

```
<Figure size 720x720 with 0 Axes>
```



The plot shows that there is a strong linear relationship between **`Bytes`** and **`Packets`** feature. The **`Action`** 'allow' is dominating the plot due to random distributions of the features. But, it can be inferred that, higher values of **`Bytes`**, **`Packets`** and **`Elapsed Time (sec)`** generally attributes to the allowed traffic. However, nothing can be said about lower values of these features as the classes overlap.

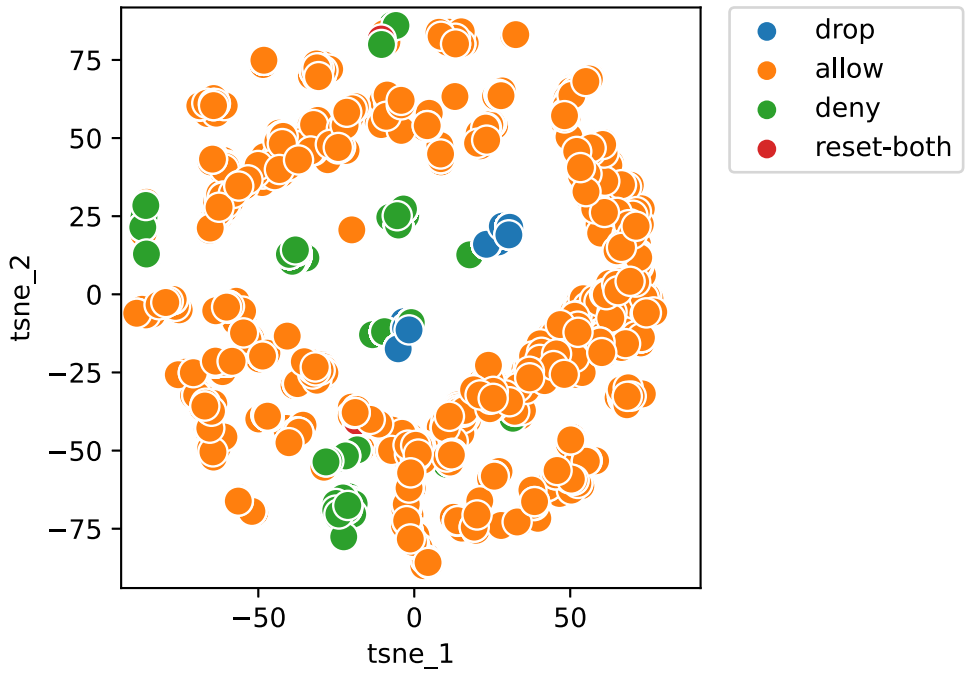
2.5. Multivariate Analysis:

We will now visualize the all numerical features in two dimentional embedded space using T-sne. This will give us a better idea about the seperability of the **`Action`** class by numerical features.

In [24]:

```
from sklearn.manifold import TSNE
from sklearn.model_selection import train_test_split

# using sampled dataset of 10000 instances with all the numerical features to limit fil
sample_df = data[['Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed Time (se
N = 10000
sample_df = sample_df.groupby('Action', group_keys=False).apply(lambda x: x.sample(int(
y = sample_df['Action']
X = sample_df.drop('Action', axis = 1)

# TSNE embedding with 2 dimensions
n_components = 2
tsne = TSNE(n_components, random_state = 859)
tsne_result = tsne.fit_transform(X)

# plotting the result of TSNE with labels referred from (https://danielmuellerkomorowsk
tsne_result_df = pd.DataFrame({'tsne_1': tsne_result[:,0], 'tsne_2': tsne_result[:,1],
fig, ax = plt.subplots(1)
sns.scatterplot(x='tsne_1', y='tsne_2', hue='label', data=tsne_result_df, ax=ax, s=120)
lim = (tsne_result.min()-5, tsne_result.max()+5)
ax.set_title("TSNE Visualization", fontdict = title_font, pad = 20.0)
ax.set_xlim(lim)
ax.set_ylim(lim)
ax.set_aspect('equal')
ax.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.0)
plt.show()
```

TSNE Visualization



The visualization backs what we have derived from earlier analysis. `Action` class 'allow' is seperable from other three `Action` classes. But, the numerical features alone are not enough to distinguish the `Action` classes. We should employ feature engineering techniques on port features for better classification.

3. Feature Engineering:

As the exploratory data analysis showed, numerical features on their own are not enough to classify all the `Action` classes. However, we can use information about port numbers involved in the traffic to engineer some features which may help us classify the `Action` classes better.

First, let us understand the port numbers in the dataset. Port numbers in network terminology refer to the numbers that are assigned to specific services that help to identify for which service each packet is intended. There are 65535 total ports available to carry out any network communication. Some of the examples are File Transfer Protocol (FTP) which uses port 20 to transfer files, Email delivery services such as Post Office Protocol (POP3)

which uses port 110 and Internet Message Access Protocol (IMAP) which uses port 143.

While port number denotes the type of service requested, it does not say anything about the host device. The host device is identified by the IP (Internet Protocol) address which is simply an address of that device on the internet. There are two types of IP addresses. Private IP address which is an IP address of the device accessing the internet assigned by the router and public IP address which is provided by Internet Service Provider (ISP) to enable the connection to the internet. The IP addresses are defined by Internet Protocol version 4 (IPv4) which is a global convention by using an unique 32-bit number. Hence, there is still a limitation on the number of unique addresses available. More specifically, there are 2^{32} unique IP addresses available.

But an important caveat is that devices not connected to the Internet, such as factory machines that communicate only with each other via TCP/IP, need not have globally unique IP addresses. These types of private networks are widely used of which our University network is a good example. There are three non overlapping ranges of nearly 18 million IP addresses reserved for these private networks. This enables the devices connected to the shared network environment to connect to internet using the same public IP address.

For this purpose, the routers or the networking hub implement a technique known as Network Address Translation (NAT). Basically, NAT allows a single device, such as a router, to act as an agent between the Internet (or public network) and a local network (or private network), which means that only a single unique IP address is required to represent an entire group of computers to anything outside their network (i. e. internet). While NAT allows for better use of IP address space, it is not always called into

action for each communication at router level. That means, sometimes port numbers can be used over the internet without translating as well.

3.1. Port Translation:

</p>

The dataset we are using has recorded port numbers on private devices as well as port numbers translated by NAT. Hence we can create two features for source and destination based on information if Port Translation (NAT) was required while passing the traffic. The features `Source Port Translation` and `Destination Port Translation` can be encoded by 1 if port numbers on devices and NAT are different indicating requirement of NAT. Otherwise, they will be encoded by 0 indicating no translation was required.

In [109…

```
data['Source Port Translation'] = (data['Source Port'] != data['NAT Source Port']).asty
data['Destination Port Translation'] = (data['Destination Port'] != data['NAT Destinati
```

In [110…

```
data.head()
```

Out[110…

	Source Port	Destination Port	NAT Source Port	NAT Destination Port	Action	Bytes	Bytes Sent	Bytes Received	Packets	Elapsed Time (sec)	pkts_se
0	57222	53	54587	53	allow	177	94	83	2	30	
1	56258	3389	56258	3389	allow	4768	1600	3168	19	17	
2	6881	50321	43265	50321	allow	238	118	120	2	1199	
3	50553	3389	50553	3389	allow	3327	1438	1889	15	17	
4	50002	443	45848	443	allow	25358	6778	18580	31	16	

We can also extract more features by building a network of source and destination ports by the help of `networkx` library. There may be an implicit importance of particular port involved in determining the `Action` of the traffic. By building a network, we can explore these features. In the dataset, we have source and destination ports on the sides of both host

devices as well as NAT. Hence, we can build two networks from these port numbers.

In [111…

```
import networkx as nx

# building bidirectional graph by using port numbers on host devices
HOST_NW = nx.DiGraph(name = "Host")
HOST_NW.add_edges_from(data[['Source Port', 'Destination Port']].values)
print(nx.info(HOST_NW))

print("-"*50)

# building bidirectional graph by using port numbers on NAT
NAT_NW = nx.DiGraph(name = "NAT")
NAT_NW.add_edges_from(data[['NAT Source Port', 'NAT Destination Port']].values)
print(nx.info(NAT_NW))
```

```
Name: Host
Type: DiGraph
Number of nodes: 24249
Number of edges: 38342
Average in degree:   1.5812
Average out degree:   1.5812
--------------------------------------------------
Name: NAT
Type: DiGraph
Number of nodes: 29555
Number of edges: 33625
Average in degree:   1.1377
Average out degree:   1.1377
```

3.2. Common Ports:

The number of common ports between Source port and the destination port can be considered a feature. Common ports for both host device and NAT can be used. Let Γ(P) denote the no. of ports connected directly to port P, also known as degree of P and P_s and P_d be the Source and destination ports respectively. Then, common ports can be calculated as-

$$CP(P_s, P_d) = \left| \Gamma(P_s) \bigcap \Gamma(P_d) \right| \tag{1}$$

In [112…

```
def common_ports(nw, src, dst):
    """
    Counts no. of common ports connected directly between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port
```

```
    Returns:
        No. of common ports from intersection of set of neighbors of both src and dst p
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst))))
    except:
        return 0

# adding features to the dataset
data['Host CP'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT CP'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro
```

3.3. Jaccard Index:

Jaccard index is calculated by number of common ports between Source port and the destination port divided by total no. of ports connected to each of the source and destination ports.

$$JI(P_s, P_d) = \frac{|\Gamma(P_s) \bigcap \Gamma(P_d)|}{|\Gamma(P_s) \bigcup \Gamma(P_d)|} \tag{2}$$

In [113…

```
def jaccard_index(nw, src, dst):
    """
    Counts Jaccard index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Jaccard Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / len(s
    except:
        return 0

# adding features to the dataset
data['Host JI'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT JI'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro
```

3.4. Salton Index:

Salton Index is also known as cosine similarity. It calculates the ratio of the number of shared ports between Source and destination port to the square root of product of their degrees.

$$SL(P_s, P_d) = \frac{|\Gamma(P_s) \bigcap \Gamma(P_d)|}{\sqrt{|\Gamma(P_s)| . |\Gamma(P_d)|}} \tag{3}$$

In [114…

```
def salton_index(nw, src, dst):
    """
    Counts Salton index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Salton Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / np.sq
    except:
        return 0

# adding features to the dataset
data['Host SL'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT SL'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro
```

3.5. Sorensen Index:

Sorensen Index is similarr to Jaccard index except the denominator is net sum in contrast to union.

$$SI(P_s, P_d) = \frac{|\,\Gamma(P_s) \bigcap \Gamma(P_d)\,|}{|\,\Gamma(P_s)| + |\Gamma(P_d)\,|} \tag{4}$$

In [115…

```
def sorensen_index(nw, src, dst):
    """
    Counts Sorensen index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Sorensen Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / (len(
    except:
        return 0

# adding features to the dataset
data['Host SI'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT SI'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro
```

3.6. Adamic-Adar Index:

Adamic-Adar Index is specifically designed for comparison of two web pages. It is similar to common ports but it penalizes rare web pages by taking log which in our case may be fraudulent ones.

$$AA(P_s, P_d) = \sum_{P \in |(\Gamma(P_s) \bigcap \Gamma(P_d))|} \frac{1}{|log(\Gamma(P)|} \tag{5}$$

In [116…

```
def adamic_adar_index(nw, src, dst):
    """
    Counts Adamic-Adar index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Adamic-Adar Index
    """
    try:
        ports = set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))
        return 1/np.sum([np.log10(set(nw.neighbors(port))) for port in ports])
    except:
        return 0

# adding features to the dataset
data['Host AA'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT AA'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro
```

3.7. Page Rank:

PageRank computes a ranking of the ports in the network graph G based on the structure of the incoming links. It was originally designed as an algorithm to rank web pages in Google search engine. We can calculate page ranks for both Source and Destination ports from Host and NAT networks.

In [117…

```
# networkx library provides a function to calculate page rank of ports in the networks.
host_page_rank = nx.pagerank(HOST_NW)
nat_page_rank = nx.pagerank(NAT_NW)

# adding features to the dataset
data['Host Source PR'] = data.apply(lambda row: host_page_rank.get(row['Source Port'],
data['Host Destination PR'] = data.apply(lambda row: host_page_rank.get(row['Destinatio
data['NAT Source PR'] = data.apply(lambda row: nat_page_rank.get(row['NAT Source Port']
data['NAT Destination PR'] = data.apply(lambda row: nat_page_rank.get(row['NAT Destinat
```

4. Data Preprocessing:

4.1. Splitting the dataset in training and validation sets:

We now have the dataset to build the machine learning models. But, as we don't have seperate test set, we cannot evaluate the performance of our model. Hence, we will split our dataset in training set and validation set using stratify sampling technique to maintain the same distribution of `Action` classes in both the sets. We will also hold out one set as test set to evaluate the logloss.

In [118…

```
from sklearn.model_selection import train_test_split
y = data['Action']
X = data.drop(['Action'], axis = 1)
X_train, X_cv, y_train, y_cv = train_test_split(X, y, test_size = 0.10, stratify = y, r
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size = 0.20,
print(f"The shape of the training set: {X_train.shape}")
print(f"The shape of the validation set: {X_cv.shape}")
print(f"The shape of the test set: {X_test.shape}")
```

```
The shape of the training set: (45972, 27)
The shape of the validation set: (6385, 27)
The shape of the test set: (11493, 27)
```

4.2. Scaling the values:

As we have seen earlier, the numerical values in the dataset have very different scales. Hence, we need to scale the those features. The RobustScaler from the sklearn library will be a good fit to scale these numerical features as it nullifies the effects of outliers in the dataset. For engineered features we can use simple StandardScaler.

In [119…

```
from sklearn.preprocessing import StandardScaler, RobustScaler

# applying RobustScaler
robust_scaler = RobustScaler()
robust_scaler_features = ['Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed
X_train_robust_scaled = robust_scaler.fit_transform(X_train[robust_scaler_features])
X_cv_robust_scaled = robust_scaler.transform(X_cv[robust_scaler_features])
X_test_robust_scaled = robust_scaler.transform(X_test[robust_scaler_features])

# applying StandardScaler
std_scaler = StandardScaler()
std_scaler_features = ['Host CP', 'NAT CP', 'Host JI', 'NAT JI', 'Host SL', 'NAT SL', '
X_train_std_scaled = std_scaler.fit_transform(X_train[std_scaler_features])
X_cv_std_scaled = std_scaler.transform(X_cv[std_scaler_features])
X_test_std_scaled = std_scaler.transform(X_test[std_scaler_features])

# stacking the scaled features
```

```
X_train_preprocessed = np.hstack((X_train[X_train.columns[:4]], X_train_robust_scaled,
X_cv_preprocessed = np.hstack((X_cv[X_cv.columns[:4]], X_cv_robust_scaled, X_cv_std_sca
X_test_preprocessed = np.hstack((X_test[X_test.columns[:4]], X_test_robust_scaled, X_te

# checking the dimensions of the datasets after preprocessing
print(f"The shape of the training set after preprocessing: {X_train_preprocessed.shape}
print(f"The shape of the validation set after preprocessing: {X_cv_preprocessed.shape}"
print(f"The shape of the test set after preprocessing: {X_test_preprocessed.shape}")
```

```
The shape of the training set after preprocessing: (45972, 27)
The shape of the validation set after preprocessing: (6385, 27)
The shape of the test set after preprocessing: (11493, 27)
```

5. Data Modelling:

5.1. Choosing the performance metric:

To evaluate the performance of any machine learning model, choosing the right metric is very important. The model is only as good as the metric shows it to be. But, the metric itself does not guarantee that the model will work as good in real-time. Hence, for any underlying task, the metric must be suitable for it's application in real-time.

The task here, is to classify **`Action`** class of a firewall log data. This is a multiclass classification task as **`Action`** class belongs to four distinct classes. The distribution among those classes is also unequal which signifies that this is an imbalanced dataset. Hence, simple accuracy or AUC cannot be used as it does not take class imbalance into account. Also, predicting every class correctly is equally as important. Moreover the model implemented should be able to predict probability of the predicted class for interpretability rather than hard constrained predictions of classes directly.

That leaves us with multiclass logarithmic loss as the metric. Typically, even if it is interpreted as a measure of error by the model, we can use it as metric to evaluate the model in a sense that the performance of the model with lower log-loss is considered better. This is because it penalizes predictions which are not confident. It is defined as-

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

$$\begin{aligned}\text{Where, } N &= \text{number of instances}\\ M &= \text{the number of classes}\\ y_{ij} &= \text{class label of } i^{th} \text{ instance for } j^{th} \text{ class}\\ p_{ij} &= \text{probability of } i^{th} \text{ instance belonging to } j^{th} \text{ class}\end{aligned}$$

As we only have four **`Action`** classes, we can also visualize the performance of the model using confusion matrix for more interpretability.

In [120…

```
def plot_confusion_matrix(truths, predictions, labels = data['Action'].unique()):
    """
    Plots the confusion matrix for all the classes in order to interpret the performanc

    Args:
        truths: Actual labels of the instances
        predictions: predicted labels of the instances by the model

    Returns:
        None
    """
    print("\n")
    matrix = confusion_matrix(truths, predictions)
    # matrix = (4,4) matrix where each cell (i,j) represents number of points of class
    precision = matrix/np.sum(matrix, axis = 0)
    # precision = (4,4) matrix where each cell (i,j) represents it's precision i.e. TP/
    recall = (matrix.T/np.sum(matrix, axis = 1)).T
    # recall = (4,4) matrix where each cell (i,j) represents it's recall i.e. TP/(TP+FN

    # creating axes to draw plots
    fig, ax = plt.subplots(1, 3)
    cmap = sns.light_palette("purple")

    # plotting the matrices using heatmaps from seaborn library
    titles = [("Confusion matrix", matrix), ("Precision matrix", precision), ("Recall m
    for i, j in enumerate(titles):
        if j[0] == "Confusion matrix":
            sns.heatmap(j[1], annot = True, cmap = cmap, fmt = "", xticklabels = labels
        else:
            sns.heatmap(j[1], annot = True, cmap = cmap, vmin = 0, vmax = 1, fmt = ".2f
        ax[i].set_xticklabels(ax[i].get_xmajorticklabels())
        ax[i].set_yticklabels(ax[i].get_ymajorticklabels(), rotation = 0)
        ax[i].set_title(j[0], fontdict = title_font)
        ax[i].set_xlabel("Predicted labels", fontdict = label_font)
        ax[i].set_ylabel("Original labels", fontdict = label_font)

    # rescaling the figure
    fig.set_figheight(5)
    fig.set_figwidth(15)
    fig.tight_layout()
```

5.2. Baseline Model:

The performance metric which we have chosen (i.e. multiclass log-loss) is not interpretable on it's own as it is just a number. Hence, to benchmark our model we must get a sense of log-loss of a random baseline model. As the **`Action`** class in our dataset has unequal distribution, we can build a random model which predicts the **`Action`** class by generating random probability of each **`Action`** class proportional to it's weight in the dataset.

In [121…

```
def baseline_model(feature_matrix, train_labels):
    """
    Generates random probabilities of output class proportional to their weights, where
    of instances in the train data

    Args:
        feature_matrix: Feature matrix of the dataset
        train_labels: Actual labels of the instances in the train data

    Returns:
        random class labels as predictions
    """
    classes = list(train_labels.unique())
    # classes = no.of unique classes
    class_weights = [np.sum(train_labels == i)/train_labels.shape[0] for i in train_lab
    # class_weights = list of class weights computed from train data
    labels = np.random.choice(classes, size = feature_matrix.shape[0], p = class_weight
    # labels = predicted labels by random sampling of class labels proportional to thei
    probs = np.zeros((labels.shape[0], len(classes)))
    # probs = matrix of predicted probabilities
    for i in range(labels.shape[0]):
        # generate probabilities by adding random numbers between 0 and 1 to predicted
        probs[i] = classes.index(labels[i]) + np.random.random_sample(len(classes))
        # normalizing probabilities of all classes so that they add up to 1
        probs[i] /= np.sum(probs[i])
    # probs = normalized random probabilities of output classes
    return labels, probs
```

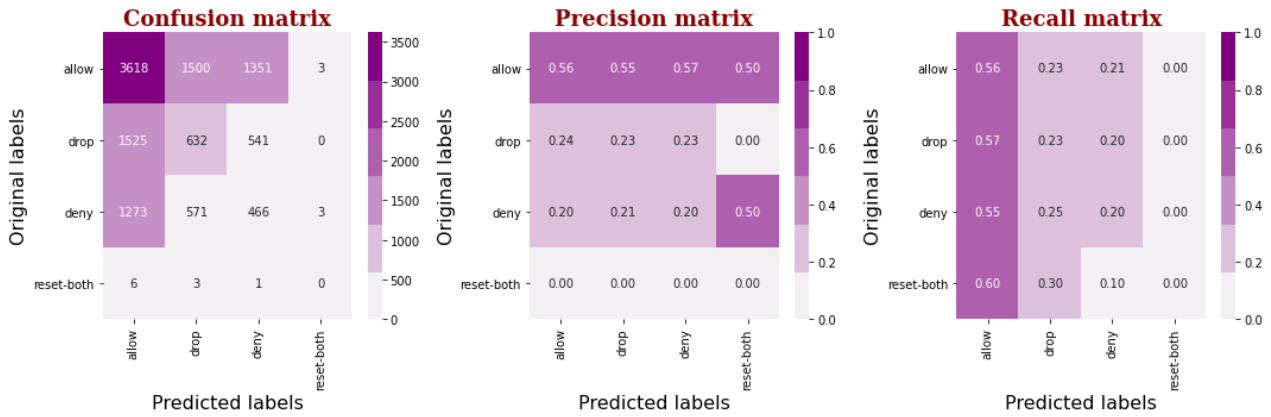
We can now check for the performance of the baseline model by calculating log-loss on the test data. We can also visualize the performance of the baseline model by plotting confusion, precision and recall matrices.

In [122…

```
from sklearn.metrics import log_loss, confusion_matrix
random_model_train_labels, random_model_train_probs = baseline_model(X_train, y_train)
random_model_test_labels, random_model_test_probs = baseline_model(X_test, y_train)

random_model_train_loss = log_loss(y_train, random_model_train_probs, eps=1e-15)
print(f"Log loss on Train Data using Random Model: {random_model_train_loss}")
random_model_test_loss = log_loss(y_test, random_model_test_probs, eps=1e-15)
print(f"Log loss on Test Data using Random Model: {random_model_test_loss}")
plot_confusion_matrix(y_test, random_model_test_labels)
```

```
Log loss on Train Data using Random Model: 1.4455086549570748
```

```
Log loss on Test Data using Random Model: 1.4488972396052735
```



The baseline model has log-loss of 1.45. Hence, in order to be able to productionize our model, we have to get log-loss well below 1.45.

5.3. K-Neighbours Classifier:

Now, let us proceed to modelling. We can start by `KNeighborsClassifier` which is one of the simplest machine learning models around. It generates the output probabilities by taking into account labels of it's nearest k neighbors by virtue of distances. We can find this optimal k value by using `RandomizedSearchCV` hyperparameter tuning technique.

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import RandomizedSearchCV

# tuning the hyperparameters on training data
knn_parameters = {'n_neighbors': [5, 8, 10, 15, 20], 'weights': ['uniform', 'distance']
knn = KNeighborsClassifier()
random_search_knn = RandomizedSearchCV(estimator = knn, param_distributions = knn_param
random_search_knn.fit(X_train_preprocessed, y_train)
print(random_search_knn.best_params_)
```

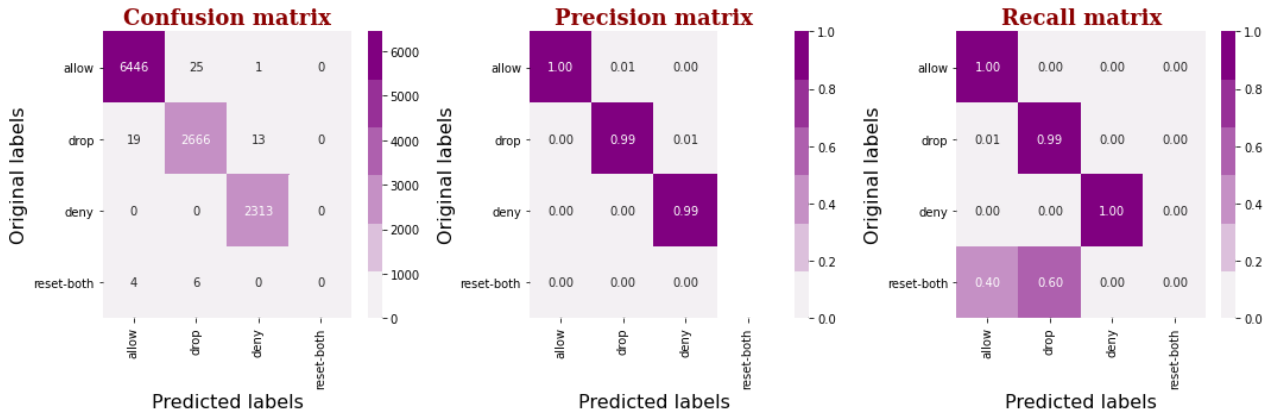
```
{'weights': 'distance', 'n_neighbors': 8}
```

Now, we have the optimal parameters to train the KNearest Neighbors model. We will also calibrate the output probabilities using CalibratedSearchCV on validation set. For evaluation, we can calculate log-loss from test data and also visualize confusion matrix on test data.

In [124…
```
# training the classifier on training data
```

```
knn = KNeighborsClassifier(n_neighbors = 8, weights = 'distance')
knn.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_knn = CalibratedClassifierCV(knn, method = 'isotonic', cv = 'prefit')
calibrator_knn.fit(X_cv_preprocessed, y_cv)

# evaluating the classifier on test data
y_train_probs = calibrator_knn.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_knn.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_knn.predict_proba(X_test_preprocessed)

train_loss_knn = log_loss(y_train, y_train_probs)
cv_loss_knn = log_loss(y_cv, y_cv_probs)
test_loss_knn = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using K-Nearest Neighbors Model: {train_loss_knn}")
print(f"Log loss on Validation Data using K-Nearest Neighbors Model: {cv_loss_knn}")
print(f"Log loss on Test Data using K-Nearest Neighbors Model: {test_loss_knn}")

y_test_preds = calibrator_knn.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using K-Nearest Neighbors Model: 0.004910136365194158
Log loss on Validation Data using K-Nearest Neighbors Model: 0.029931167578385855
Log loss on Test Data using K-Nearest Neighbors Model: 0.028648821088319024
```



As we can see, the model has test log-loss of 0.028 which is very low compared to 1.45 of the baseline model. But, even if the model is doing very well in classifying among all the **`Action`** classes except 'reset-both'. It may be struggling because there are only 10 instances of that class in the test data.

5.4. Logistic Regression:

The **`LogisticRegression`** is a classification model which explicitly tries to minimize the log-loss. The parameter 'C' controls the amount of regularization with smaller values having more regularization. 'penalty' parameter controls the type of regularization.

In [ ]:

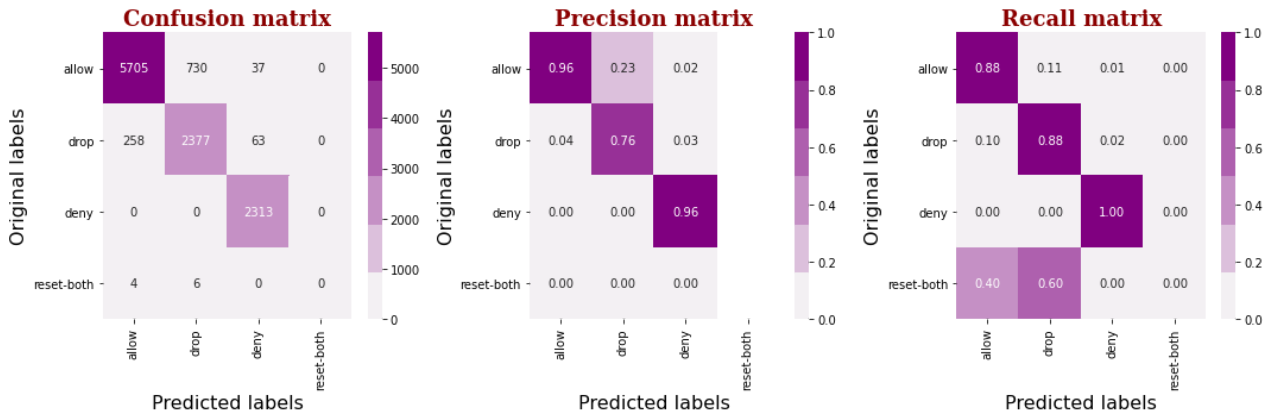
```
from sklearn.linear_model import LogisticRegression

# tuning the hyperparameters on training data
log_parameters = {'C': [0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}
log = LogisticRegression(class_weight = 'balanced', n_jobs = -1, random_state = 859)
random_search_log = RandomizedSearchCV(estimator = log, param_distributions = log_param
random_search_log.fit(X_train_preprocessed, y_train)
print(random_search_log.best_params_)
```

```
{'penalty': 'l2', 'C': 1}
```

In [125…

```
# training the classifier on training data
logistic = LogisticRegression(C = 1, penalty = 'l2', class_weight = 'balanced', n_jobs
logistic.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_logistic = CalibratedClassifierCV(logistic, method = 'isotonic', cv = 'prefi
calibrator_logistic.fit(X_cv_preprocessed, y_cv)

# # evaluating the classifier on test data
y_train_probs = calibrator_logistic.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_logistic.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_logistic.predict_proba(X_test_preprocessed)

train_loss_logistic = log_loss(y_train, y_train_probs, eps=1e-15)
cv_loss_logistic = log_loss(y_cv, y_cv_probs, eps=1e-15)
test_loss_logistic = log_loss(y_test, y_test_probs, eps=1e-15)

print(f"Log loss on Train Data using Logistic Regression Model: {train_loss_logistic}")
print(f"Log loss on Validation Data using Logistic Regression Model: {cv_loss_logistic}
print(f"Log loss on Test Data using Logistic Regression Model: {test_loss_logistic}")

y_test_preds = calibrator_logistic.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using Logistic Regression Model: 0.3741446688468921
Log loss on Validation Data using Logistic Regression Model: 0.3689663502311728
Log loss on Test Data using Logistic Regression Model: 0.373433355390849
```



As we can see, the model has test log-loss of 0.38 which is low compared to 1.45 of the baseline model but higher than 0.028 of K-neighbors

model. Also, similar to k-neighbors, the model is doing very well in classifying among all the `Action` classes except 'reset-both'. It may be struggling because there are only 10 instances of that class in the test data.

5.5. Support Vector Classifier:

The `SGDClassifier` is a classification model which explicitly tries to minimize the hinge-loss. Hence, it is primarily a supprot vector classifier with linear kerenel. We are not using supprot vectors with rbf kernel because of high number of training samples. The parameter 'alpha' controls the amount of regularization with smaller values having less regularization. 'penalty' parameter controls the type of regularization.

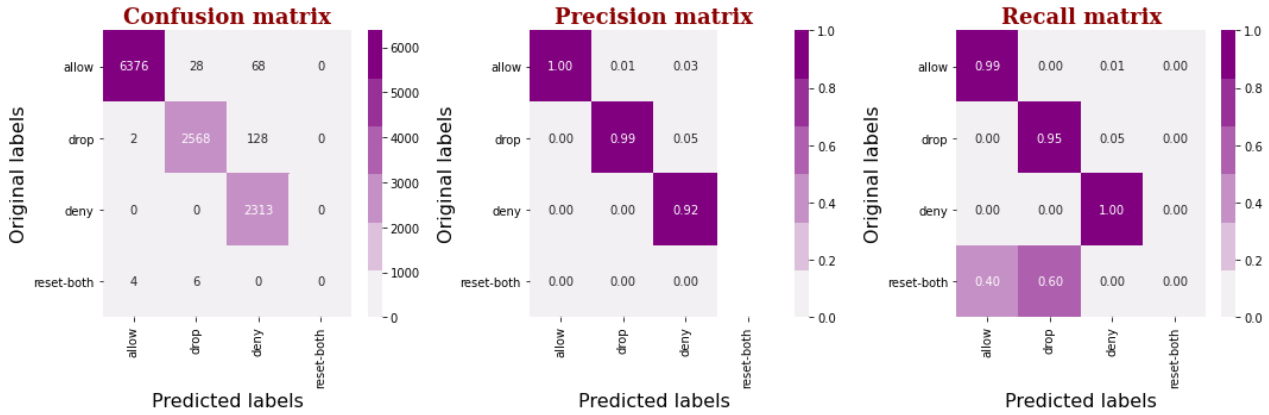
In [ ]:
```
from sklearn.linear_model import SGDClassifier

# tuning the hyperparameters on training data
svc_parameters = {'alpha': [0.0001, 0.01, 1, 100, 10000], 'penalty': ['l1', 'l2']}
svc = SGDClassifier(class_weight = 'balanced', n_jobs = -1, random_state = 859)
random_search_svc = RandomizedSearchCV(estimator = svc, param_distributions = svc_param
random_search_svc.fit(X_train_preprocessed, y_train)
print(random_search_svc.best_params_)
```

```
{'penalty': 'l1', 'alpha': 1}
```

In [126…
```
# training the classifier on training data
svc = SGDClassifier(alpha = 1, penalty = 'l1', class_weight = 'balanced', n_jobs = -1,
svc.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_svc = CalibratedClassifierCV(svc, method = 'isotonic', cv = 'prefit')
calibrator_svc.fit(X_cv_preprocessed, y_cv)

# calibrating the classifier on test data
y_train_probs = calibrator_svc.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_svc.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_svc.predict_proba(X_test_preprocessed)

train_loss_svc = log_loss(y_train, y_train_probs)
cv_loss_svc = log_loss(y_cv, y_cv_probs)
test_loss_svc = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using Support Vectors Model: {train_loss_svc}")
print(f"Log loss on Validation Data using Support Vectors Model: {cv_loss_svc}")
print(f"Log loss on Test Data using Support Vectors Model: {test_loss_svc}")

y_test_preds = calibrator_svc.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using Support Vectors Model: 0.08837143334849633
Log loss on Validation Data using Support Vectors Model: 0.0698077448284138
```

```
Log loss on Test Data using Support Vectors Model: 0.08825439909360266
```



As we can see, the model has test log-loss of 0.09 which is low compared to 1.45 of the baseline model but higher than 0.028 of K-neighbors model. Also, similar to k-neighbors, the model is doing very well in classifying among all the `Action` classes except 'reset-both'. It may be struggling because there are only 10 instances of that class in the test data.

5.6. Random Forest Classifier:

The `RandomForestClassifier` is an ensemble model which trains large ensemble of decision trees. The parameter 'n_estimators' controls the amount of decision tree to train. 'criterion' controls the information gain metric on which to split the tree. 'max_depth' is the maximum no. of levels of trees allowed in construction of decision trees.

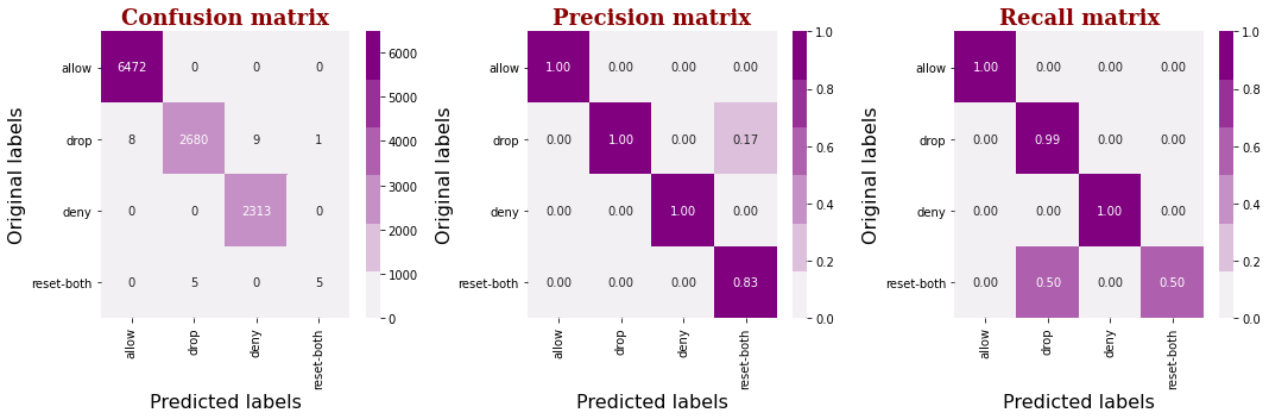
In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

# tuning the hyperparameters on training data
rf_parameters = {'n_estimators': [100, 500, 1000, 2000], 'criterion': ['gini', 'entropy
rf = RandomForestClassifier(class_weight = 'balanced', n_jobs = -1, random_state = 859)
random_search_rf = RandomizedSearchCV(estimator = rf, param_distributions = rf_paramete
random_search_rf.fit(X_train_preprocessed, y_train)
print(random_search_rf.best_params_)
```

```
{'n_estimators': 2000, 'max_depth': 20, 'criterion': 'entropy'}
```

In [127…
```
# training the classifier on training data
rf = RandomForestClassifier(n_estimators = 2000, max_depth = 20, criterion = 'entropy',
rf.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_rf = CalibratedClassifierCV(rf, method = 'isotonic', cv = 'prefit')
calibrator_rf.fit(X_cv_preprocessed, y_cv)
```

```
# evaluating the classifier on test data
y_train_probs = calibrator_rf.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_rf.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_rf.predict_proba(X_test_preprocessed)

train_loss_rf = log_loss(y_train, y_train_probs)
cv_loss_rf = log_loss(y_cv, y_cv_probs)
test_loss_rf = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using Random Forest Classifier Model: {train_loss_rf}")
print(f"Log loss on Validation Data using Random Forest Classifier Model: {cv_loss_rf}"
print(f"Log loss on Test Data using Random Forest Classifier Model: {test_loss_rf}")

y_test_preds = calibrator_rf.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using Random Forest Classifier Model: 0.004069154024901952
Log loss on Validation Data using Random Forest Classifier Model: 0.008991514799078594
Log loss on Test Data using Random Forest Classifier Model: 0.00923073679134964
```

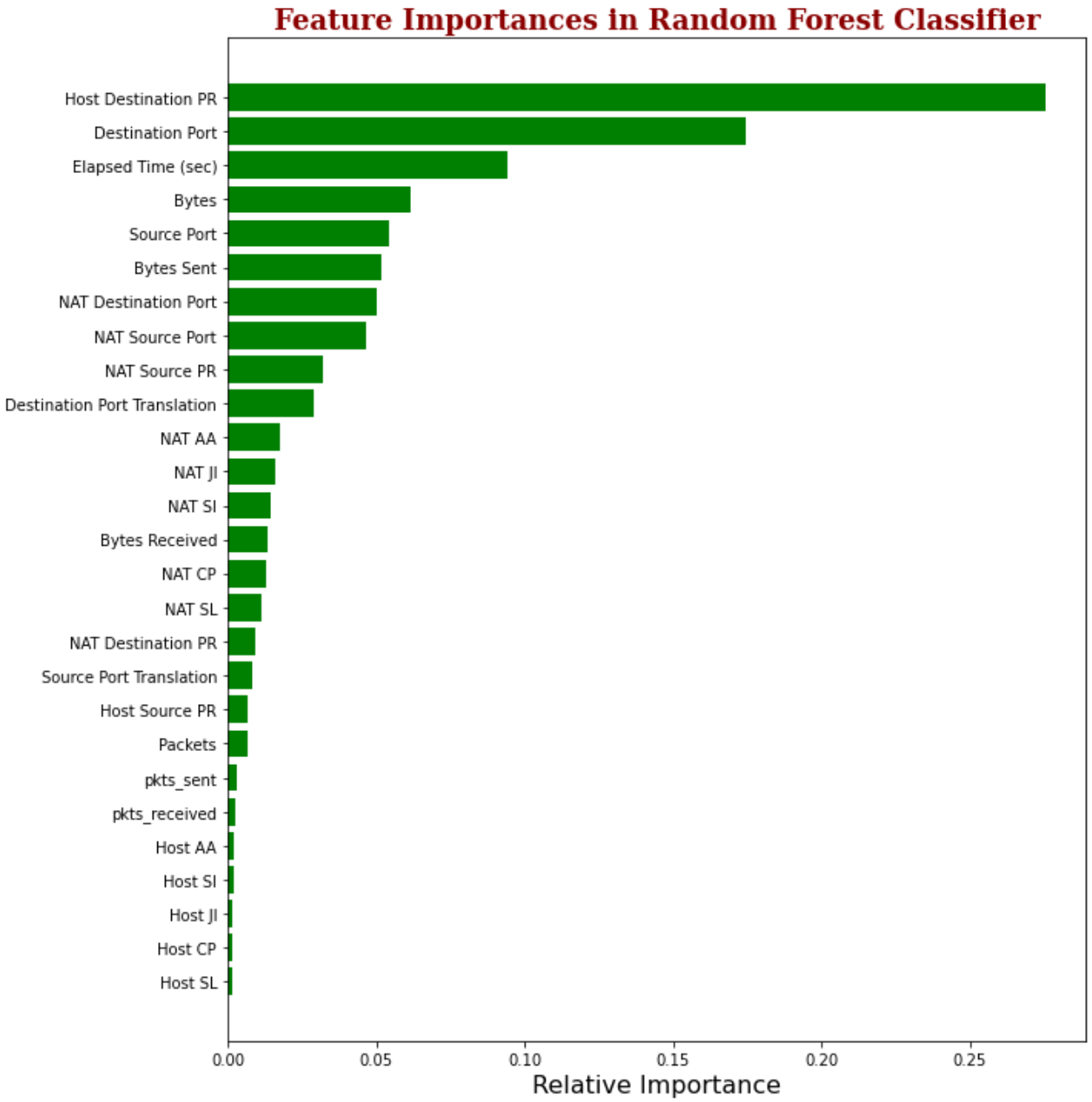


As we can see, the model has test log-loss of 0.009 which is very low compared to 1.45 of the baseline model and also 0.028 of K-neighbors model. Also, in contarst to to k-neighbors, the model is doing very well in classifying among all the **`Action`** classes including 'reset-both'. As this seems to be a good model, let us check the importances of the features in predicting the **`Action`** class.

In [ ]:

```
# list of features
features = ['Source Port', 'Destination Port', 'NAT Source Port', 'NAT Destination Port
            'Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed Time (sec)', '
            'pkts_received', 'Host Source PR', 'Host Destination PR', 'NAT Source PR',
            'Host CP', 'NAT CP', 'Host JI', 'NAT JI', 'Host SL', 'NAT SL', 'Host SI', '
            'NAT AA', 'Source Port Translation', 'Destination Port Translation']
# array of feature importances
importances = rf.feature_importances_
# sorting indices of importances in decreasing order
indices = (np.argsort(importances))
# plotting the horizontal barplot
plt.figure(figsize=(10,12))
```

```
plt.title('Feature Importances in Random Forest Classifier', fontdict = title_font)
plt.barh(range(len(indices)), importances[indices], color='green', align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance', fontdict = label_font)
plt.show()
```



Notably, lot of the engineered features are very important in classifying the log. Namely, Page rank and Adamic-Adar index features are very important.

5.7. Light GBM Classifier:

The **`LGBMClassifier`** is also an ensemble model which trains large ensemble of decision trees. The parameter 'n_estimators' controls the amount

of decision tree to train. 'objective' specifies the learning task which is 'multiclass' in our case. 'max_depth' is the maximum no. of levels of trees allowed in construction of decision trees.

In [ ]:

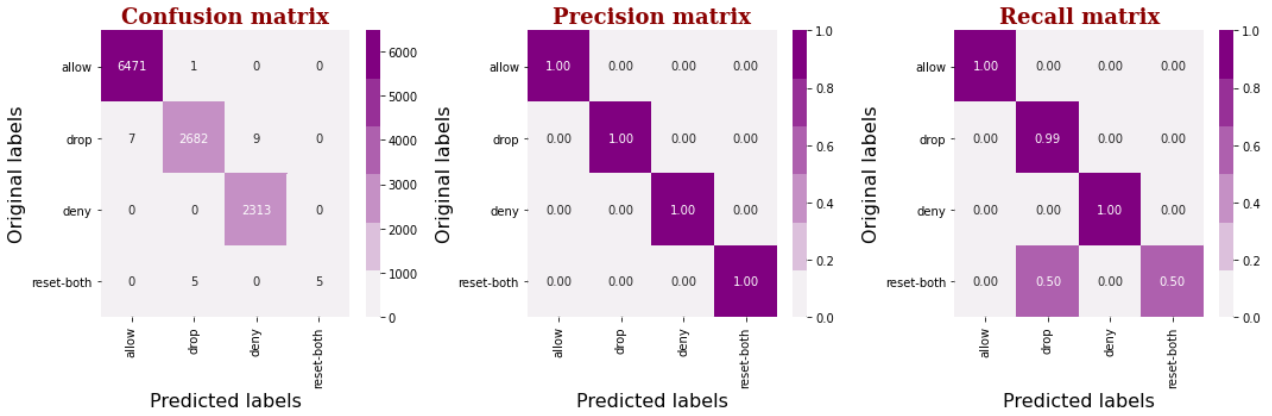
```
from lightgbm import LGBMClassifier

# tuning the hyperparameters on training data
lgbm_parameters = {'n_estimators': [500, 750, 1000, 1500], 'max_depth': [2, 4, 6, 8]}
lgbm = LGBMClassifier(objective = 'multiclass', class_weight = 'balanced', n_jobs = -1,
random_search_lgbm = RandomizedSearchCV(estimator = lgbm, param_distributions = lgbm_pa
random_search_lgbm.fit(X_train_preprocessed, y_train)
print(random_search_lgbm.best_params_)
```

```
{'n_estimators': 750, 'max_depth': 4}
```

In [128…

```
# training the classifier on training data
lgbm = LGBMClassifier(n_estimators = 750, max_depth = 4, objective = 'multiclass', clas
lgbm.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_lgbm = CalibratedClassifierCV(lgbm, method = 'isotonic', cv = 'prefit')
calibrator_lgbm.fit(X_cv_preprocessed, y_cv)

# evaluating the classifier on test data
y_train_probs = calibrator_lgbm.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_lgbm.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_lgbm.predict_proba(X_test_preprocessed)

train_loss_lgbm = log_loss(y_train, y_train_probs)
cv_loss_lgbm = log_loss(y_cv, y_cv_probs)
test_loss_lgbm = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using Light GBM Classifier Model: {train_loss_lgbm}")
print(f"Log loss on Validation Data using Light GBM Classifier Model: {cv_loss_lgbm}")
print(f"Log loss on Test Data using Light GBM Classifier Model: {test_loss_lgbm}")

y_test_preds = calibrator_lgbm.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

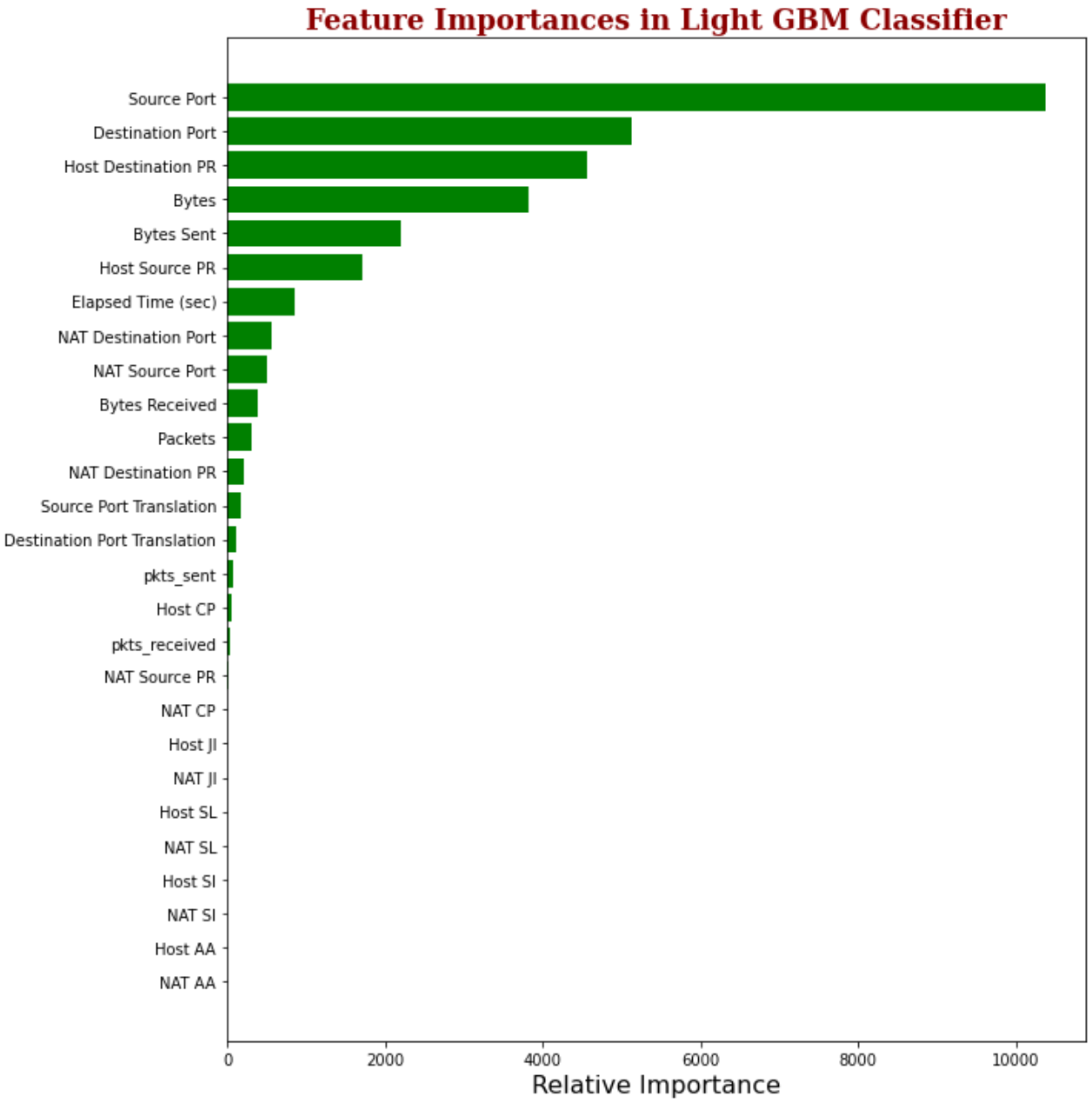
```
Log loss on Train Data using Light GBM Classifier Model: 0.003364022326366539
Log loss on Validation Data using Light GBM Classifier Model: 0.007541258733007845
Log loss on Test Data using Light GBM Classifier Model: 0.008336562890961203
```



As we can see, the model has test log-loss of 0.008 which is very low compared to 1.45 of the baseline model and also lower than 0.009 of Random forest model. Also, the model is doing very well in classifying among all the **`Action`** classes including 'reset-both'. As this seems to be the best model, let us check the importances of the features in predicting the **`Action`** class.

In [ ]:

```
# list of features
features = ['Source Port', 'Destination Port', 'NAT Source Port', 'NAT Destination Port
            'Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed Time (sec)', '
            'pkts_received', 'Host Source PR', 'Host Destination PR', 'NAT Source PR',
            'Host CP', 'NAT CP', 'Host JI', 'NAT JI', 'Host SL', 'NAT SL', 'Host SI', '
            'NAT AA', 'Source Port Translation', 'Destination Port Translation']
# array of feature importances
importances = rf.feature_importances_
# sorting indices of importances in decreasing order
indices = (np.argsort(importances))
# plotting the horizontal barplot
plt.figure(figsize=(10,12))
plt.title('Feature Importances in Light GBM Classifier', fontdict = title_font)
plt.barh(range(len(indices)), importances[indices], color='green', align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance', fontdict = label_font)
plt.show()
```



Notably, lot of the engineered features are very important in classifying the log. Namely, Page rank and translation features are very important.

5.8. Adaboost Classifier:

The `AdaboostClassifier` is also an ensemble model which trains large ensemble of decision trees. The parameter 'n_estimators' controls the amount of decision tree to train.

In [ ]:

```
from sklearn.ensemble import AdaBoostClassifier

# tuning the hyperparameters on training data
adaboost_parameters = {'n_estimators': [50, 100, 200, 500], 'learning_rate': [0.1, 0.5,
adaboost = AdaBoostClassifier(random_state = 859)
```

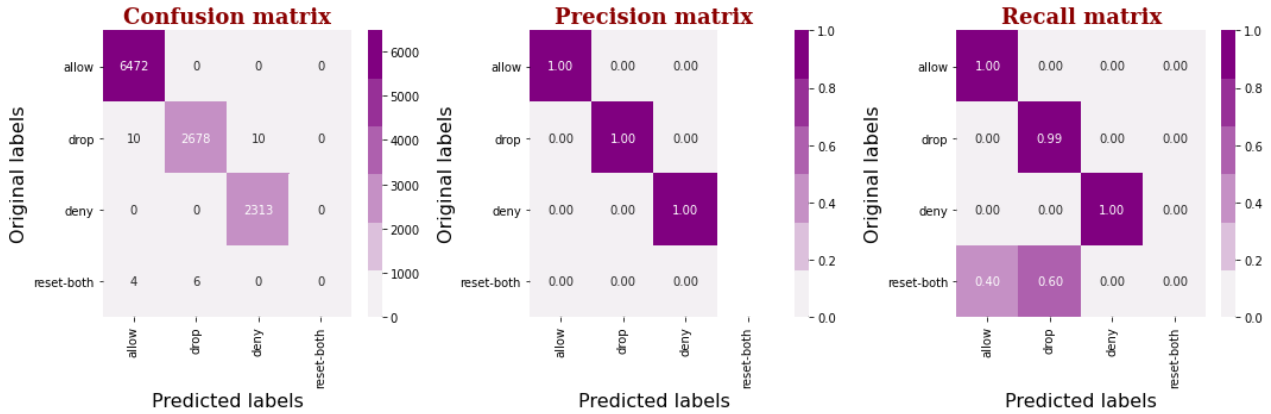
```
random_search_adaboost = RandomizedSearchCV(estimator = adaboost, param_distributions =
random_search_adaboost.fit(X_train_preprocessed, y_train)
print(random_search_adaboost.best_params_)
```

```
{'n_estimators': 50, 'learning_rate': 0.1}
```

In [129…

```
# training the classifier on training data
adaboost = AdaBoostClassifier(n_estimators = 50, learning_rate = 0.1, random_state = 85
adaboost.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_adaboost = CalibratedClassifierCV(adaboost, method = 'isotonic', cv = 'prefi
calibrator_adaboost.fit(X_cv_preprocessed, y_cv)

# evaluating the classifier on test data
y_train_probs = calibrator_adaboost.predict_proba(X_train_preprocessed)
y_cv_probs = calibrator_adaboost.predict_proba(X_cv_preprocessed)
y_test_probs = calibrator_adaboost.predict_proba(X_test_preprocessed)

train_loss_adaboost = log_loss(y_train, y_train_probs)
cv_loss_adaboost = log_loss(y_cv, y_cv_probs)
test_loss_adaboost = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using Random Forest Classifier Model: {train_loss_adaboo
print(f"Log loss on Validation Data using Random Forest Classifier Model: {cv_loss_adab
print(f"Log loss on Test Data using Random Forest Classifier Model: {test_loss_adaboost

y_test_preds = calibrator_adaboost.predict(X_test_preprocessed)
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using Random Forest Classifier Model: 0.10769807689071362
Log loss on Validation Data using Random Forest Classifier Model: 0.10445927216676672
Log loss on Test Data using Random Forest Classifier Model: 0.10796348666806237
```



As we can see, the model has test log-loss of 0.107 which is very low compared to 1.45 of the baseline model but not better than 0.008 of Light GBM model. Also, the model is doing very well in classifying among all the **`Action`** classes but is struggling to classify 'reset-both'.

5.9. Multi Layered Perceptron Classifier:

Apart from machine learning networks, we can also try with simple neural network known as Multi layered perceptron using sequential API in tensorflow library. The layer will have batch normalized input layer followed by couple of blocks of dense and dropout layers.

In [195…

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential

# designing the MLP architechture
IN_SHAPE = X_train_preprocessed.shape[-1]
NUM_CLASSES = 4
model = tf.keras.models.Sequential([
        tf.keras.layers.InputLayer(input_shape=(IN_SHAPE,), name = "Input"),
        tf.keras.layers.BatchNormalization(name = "BN"),
        tf.keras.layers.Dense(128, activation='relu', name = "Dense_1"),
        tf.keras.layers.Dropout(0.2, name = "Dropout_1"),
        tf.keras.layers.Dense(64, activation='relu', name = "Dense_2"),
        tf.keras.layers.Dropout(0.2, name = "Dropout_2"),
        tf.keras.layers.Dense(NUM_CLASSES, activation = 'softmax', name = "Output")
        ], name = "MLP_Model")
model.summary()
```

```
Model: "MLP_Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
BN (BatchNormalization)      (None, 27)                108
_________________________________________________________________
Dense_1 (Dense)              (None, 128)               3584
_________________________________________________________________
Dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
Dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
Dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
Output (Dense)               (None, 4)                 260
=================================================================
Total params: 12,208
Trainable params: 12,154
Non-trainable params: 54
_________________________________________________________________
```

To train the MLP model, the input feature matrix must be converted into float datatype and the output target vector must be converted into sparse matrix.

In [196…

```
# changing dtype of input features to float
X_train_encoded = X_train_preprocessed.astype(np.float)
X_cv_encoded = X_cv_preprocessed.astype(np.float)
X_test_encoded = X_test_preprocessed.astype(np.float)

# encoding target vectors into sparse matrices
```

```
encoder = dict(zip(np.sort(y_train.unique()), list(range(4))))
# encoder = dictionary with class labels as keys and integer as values
def encode(y):
    return encoder[y]
y_train_coded = y_train.apply(encode)
y_cv_coded = y_cv.apply(encode)
y_test_coded = y_test.apply(encode)
y_train_encoded = tf.keras.utils.to_categorical(y_train_coded)
y_cv_encoded = tf.keras.utils.to_categorical(y_cv_coded)
y_test_encoded = tf.keras.utils.to_categorical(y_test_coded)
```

In [197…

```
BATCH_SIZE = 1024
EPOCHS = 100

# designing callbacks to control training of MLP
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', factor = 0.5, pa
stopper = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience = 10)
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("MLP.h5", monitor = 'val_loss', save
def create_tensorboard_cb(model):
    import time
    import os
    root_logdir = os.path.join(os.curdir, model)
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    logdir = os.path.join(root_logdir, run_id)
    return tf.keras.callbacks.TensorBoard(logdir, histogram_freq = 1)
tensorboard_cb = create_tensorboard_cb("MLP_logs")
CALLBACKS = [reduce_lr, stopper, checkpoint_cb, tensorboard_cb]

# fitting the MLP on training data using categorical crossentropy as loss function whic
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy')
model.fit(x = X_train_encoded, y = y_train_encoded, batch_size = BATCH_SIZE, epochs = E
```

```
Epoch 1/100
45/45 [==============================] - 1s 11ms/step - loss: 0.4670 - val_loss: 0.1131
Epoch 2/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0603 - val_loss: 0.0560
Epoch 3/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0495 - val_loss: 0.0424
Epoch 4/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0428 - val_loss: 0.0365
Epoch 5/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0396 - val_loss: 0.0332
Epoch 6/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0362 - val_loss: 0.0298
Epoch 7/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0336 - val_loss: 0.0281
Epoch 8/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0318 - val_loss: 0.0268
Epoch 9/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0297 - val_loss: 0.0253
Epoch 10/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0286 - val_loss: 0.0250
Epoch 11/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0268 - val_loss: 0.0243
Epoch 12/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0260 - val_loss: 0.0232
Epoch 13/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0249 - val_loss: 0.0216
Epoch 14/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0234 - val_loss: 0.0206
Epoch 15/100
```

```
45/45 [==============================] - 0s 6ms/step - loss: 0.0223 - val_loss: 0.0193
Epoch 16/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0208 - val_loss: 0.0179
Epoch 17/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0189 - val_loss: 0.0172
Epoch 18/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0184 - val_loss: 0.0162
Epoch 19/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0178 - val_loss: 0.0154
Epoch 20/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0168 - val_loss: 0.0151
Epoch 21/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0162 - val_loss: 0.0147
Epoch 22/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0156 - val_loss: 0.0141
Epoch 23/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0153 - val_loss: 0.0142
Epoch 24/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0150 - val_loss: 0.0138
Epoch 25/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0147 - val_loss: 0.0140
Epoch 26/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0146 - val_loss: 0.0140
Epoch 27/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0144 - val_loss: 0.0140
Epoch 28/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0145 - val_loss: 0.0138
Epoch 29/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0144 - val_loss: 0.0141
Epoch 30/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0147 - val_loss: 0.0135
Epoch 31/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0139 - val_loss: 0.0133
Epoch 32/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0138 - val_loss: 0.0134
Epoch 33/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0137 - val_loss: 0.0133
Epoch 34/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0140 - val_loss: 0.0131
Epoch 35/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0139 - val_loss: 0.0136
Epoch 36/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0136 - val_loss: 0.0130
Epoch 37/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0134 - val_loss: 0.0135
Epoch 38/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0132 - val_loss: 0.0131
Epoch 39/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0129 - val_loss: 0.0129
Epoch 40/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0130 - val_loss: 0.0132
Epoch 41/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0131 - val_loss: 0.0128
Epoch 42/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0133 - val_loss: 0.0129
Epoch 43/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0130 - val_loss: 0.0132
Epoch 44/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0128 - val_loss: 0.0127
Epoch 45/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0133 - val_loss: 0.0127
Epoch 46/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0122 - val_loss: 0.0130
Epoch 47/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0129 - val_loss: 0.0127
```

```
Epoch 48/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0130 - val_loss: 0.0134
Epoch 49/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0125 - val_loss: 0.0134
Epoch 50/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0127 - val_loss: 0.0131
Epoch 51/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0124 - val_loss: 0.0128
Epoch 52/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0125 - val_loss: 0.0126
Epoch 53/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0124 - val_loss: 0.0129
Epoch 54/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0124 - val_loss: 0.0130
Epoch 55/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0120 - val_loss: 0.0129
Epoch 56/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0123 - val_loss: 0.0128
Epoch 57/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0120 - val_loss: 0.0126
Epoch 58/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0124 - val_loss: 0.0132
Epoch 59/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0120 - val_loss: 0.0127
Epoch 60/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0124 - val_loss: 0.0125
Epoch 61/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0120 - val_loss: 0.0124
Epoch 62/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0118 - val_loss: 0.0132
Epoch 63/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0121 - val_loss: 0.0126
Epoch 64/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0116 - val_loss: 0.0124
Epoch 65/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0115 - val_loss: 0.0129
Epoch 66/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0118 - val_loss: 0.0121
Epoch 67/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0117 - val_loss: 0.0133
Epoch 68/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0118 - val_loss: 0.0125
Epoch 69/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0116 - val_loss: 0.0124
Epoch 70/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0117 - val_loss: 0.0124
Epoch 71/100
45/45 [==============================] - 0s 7ms/step - loss: 0.0115 - val_loss: 0.0131
Epoch 72/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0113 - val_loss: 0.0123
Epoch 73/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0116 - val_loss: 0.0122
Epoch 74/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0114 - val_loss: 0.0125
Epoch 75/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0113 - val_loss: 0.0124
Epoch 76/100
45/45 [==============================] - 0s 6ms/step - loss: 0.0109 - val_loss: 0.0124
```

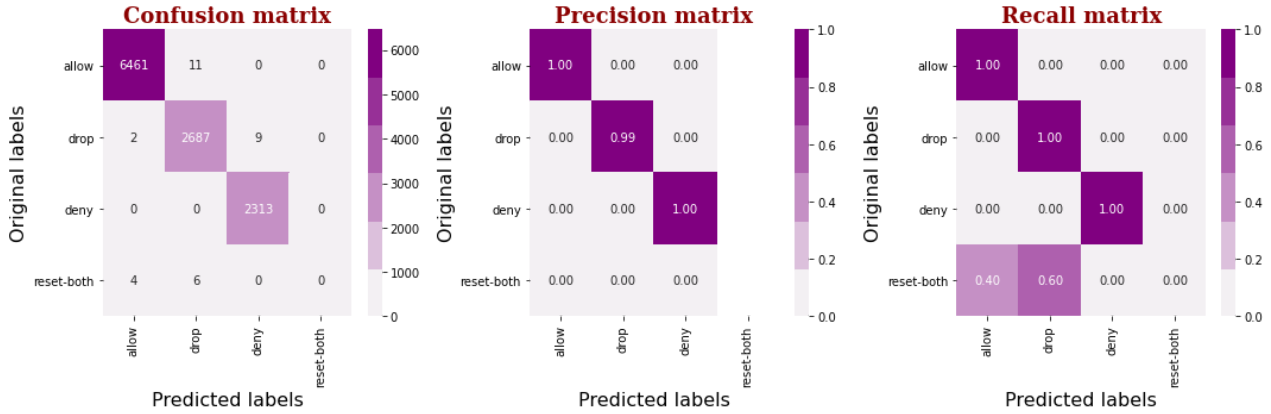
Out[197… `<tensorflow.python.keras.callbacks.History at 0x7f15b0ae2110>`

In [198…

```
# evaluating MLP on test data
y_train_probs = model.predict(X_train_encoded)
y_cv_probs = model.predict(X_cv_encoded)
```

```
y_test_probs = model.predict(X_test_encoded)
train_loss_mlp = log_loss(y_train, y_train_probs)
cv_loss_mlp = log_loss(y_cv, y_cv_probs)
test_loss_mlp = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using MLP Model: {train_loss_mlp}")
print(f"Log loss on Validation Data using Random MLP Model: {cv_loss_mlp}")
print(f"Log loss on Test Data using MLP Model: {test_loss_mlp}")

decoder = dict(zip(list(range(4)), np.sort(y_train.unique())))
# decoder = dictionary with integers as keys and class label as values
y_test_preds_indices = np.argmax(y_test_probs, axis = 1)
# predictions from MLP are probabilities of particular class labels hence, getting clas
y_test_preds = np.array([decoder[i] for i in y_test_preds_indices])
# getting class labels using decoder dictionary
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using MLP Model: 0.010249493232695911
Log loss on Validation Data using Random MLP Model: 0.012355825795534769
Log loss on Test Data using MLP Model: 0.011194943536576168
```



As we can see, the model has test log-loss of 0.011 which is very low compared to 1.45 of the baseline model but not better than 0.008 of Light GBM model. Also, the model is doing very well in classifying among all the **`Action`** classes but is struggling to classify 'reset-both'.

5.10. Stacking Classifier:

The **`StackingClassifier`** is an ensemble learning technique to combine multiple classification models via a meta-classifier. We can use well performing models like KNeighbours, RandomForest, LightGBM and Adaboost as base models and LogisticRegression as meta-classifier to minimize the log-loss. The hyperparameters of the **`StackingClassifier`** can be tuned by using RandomizedSearchCV technique.

In [199…

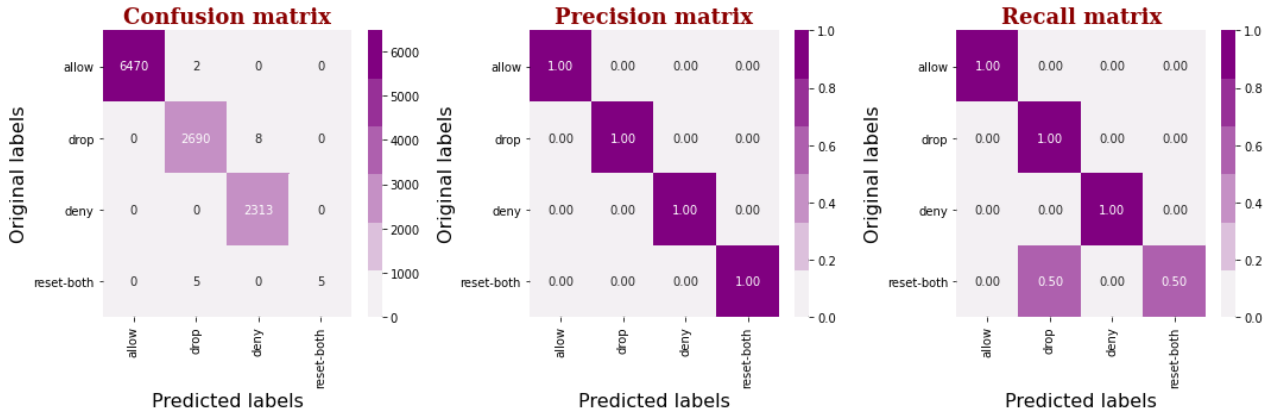
```
from mlxtend.classifier import StackingClassifier
clf1 = KNeighborsClassifier()
clf2 = RandomForestClassifier(random_state = 859)
clf3 = LGBMClassifier(random_state = 859)
clf4 = AdaBoostClassifier(random_state = 859)
lr = LogisticRegression()
stacking = StackingClassifier(classifiers = [clf1, clf2, clf3, clf4],
                              meta_classifier = lr)

stacking_parameters = {'kneighborsclassifier__n_neighbors': [2, 4, 6, 8],
          'randomforestclassifier__n_estimators': [100, 200, 500, 800],
          'randomforestclassifier__max_depth': [2, 5, 10, 15],
          'lgbmclassifier__n_estimators': [50, 100, 200, 500],
          'adaboostclassifier__n_estimators': [20, 50, 80, 100],
          'adaboostclassifier__learning_rate': [0.1, 0.5, 1],
          'meta_classifier__C': [0.1, 1, 10]}

random_search_stacking = RandomizedSearchCV(estimator = stacking, param_distributions =
random_search_stacking.fit(X_train_preprocessed, y_train_coded)
print(random_search_stacking.best_params_)
```

```
{'randomforestclassifier__n_estimators': 800, 'randomforestclassifier__max_depth': 10,
'meta_classifier__C': 10, 'lgbmclassifier__n_estimators': 500, 'kneighborsclassifier__n_
neighbors': 8, 'adaboostclassifier__n_estimators': 50, 'adaboostclassifier__learning_rat
e': 0.1}
```

In [200…

```
# training the classifier on training data
clf1 = KNeighborsClassifier(n_neighbors = 8)
clf2 = RandomForestClassifier(n_estimators = 800, max_depth = 10, random_state = 859)
clf3 = LGBMClassifier(n_estimators = 500, random_state = 859)
clf4 = AdaBoostClassifier(n_estimators = 50, learning_rate = 0.1, random_state = 859)
lr = LogisticRegression(C = 10, random_state = 859)
stacking = StackingClassifier(classifiers = [clf1, clf2, clf3, clf4],
                              meta_classifier = lr)
stacking.fit(X_train_preprocessed, y_train_coded)

# evaluating the classifier on test data
y_train_probs = stacking.predict_proba(X_train_preprocessed)
y_cv_probs = stacking.predict_proba(X_cv_preprocessed)
y_test_probs = stacking.predict_proba(X_test_preprocessed)

train_loss_stacking = log_loss(y_train, y_train_probs)
cv_loss_stacking = log_loss(y_cv, y_cv_probs)
test_loss_stacking = log_loss(y_test, y_test_probs)

print(f"Log loss on Train Data using Stacking Classifier Model: {train_loss_stacking}")
print(f"Log loss on Validation Data using Stacking Classifier Model: {cv_loss_stacking}
print(f"Log loss on Test Data using Stacking Classifier Model: {test_loss_stacking}")

y_test_preds_indices = stacking.predict(X_test_preprocessed)
# getting class labels using decoder dictionary
y_test_preds = np.array([decoder[i] for i in y_test_preds_indices])
plot_confusion_matrix(y_test, y_test_preds)
```

```
Log loss on Train Data using Stacking Classifier Model: 0.004953851483601359
Log loss on Validation Data using Stacking Classifier Model: 0.014535186771326392
Log loss on Test Data using Stacking Classifier Model: 0.010798456552344996
```



As we can see, the model has test log-loss of 0.0107 which is very low compared to 1.45 of the baseline model but not better than 0.008 of Light GBM model. Also, the model is doing very well in classifying among all the **`Action`** classes including 'reset-both'. Hence, the better model can be chosen by computing run-time complexity.

5.8. Summary of the performance of models:

We can now compare the performance of all the models on attributes such as train loss, test loss and run time in predicting the **`Action`** class.

In [201…

```
import time
def check(model, vector):
    """
    Checks the run-time complexity of the model for a single log data. Gives mean of 10

    Args:
        model: Instance of the ML model
        vector: vector of log data

    Returns:
        Mean run time of prediction
    """
    times = []
    for i in range(100):
        t = time.time()
        pred = model.predict(vector)
        times.append(time.time() - t)
    return np.round(np.mean(times), 4)

# storing run times for all the models in dictionary
elapsed_times= {}
elapsed_times['KNeighbors Classifier'] = check(calibrator_knn, X_test_preprocessed[0].r
elapsed_times['Logistic Regression'] = check(logistic, X_test_preprocessed[0].reshape(1
elapsed_times['Support Vectors'] = check(calibrator_svc, X_test_preprocessed[0].reshape
elapsed_times['Random Forest Classifier'] = check(calibrator_rf, X_test_preprocessed[0]
elapsed_times['Light GBM Classifier'] = check(calibrator_lgbm, X_test_preprocessed[0].r
elapsed_times['Adaboost Classifier'] = check(calibrator_adaboost, X_test_preprocessed[0
```

```
elapsed_times['MLP Classifier'] = check(model, X_test_encoded[0].reshape(1, -1))
elapsed_times['Stacking Classifier'] = check(stacking, X_test_preprocessed[0].reshape(1
```

In [202…

```
from prettytable import PrettyTable

# summarizing the results in table
x = PrettyTable()

x.field_names = ["Sr. No.", "Classifier", "Train Log-Loss", "Test Log-Loss", "Run-time

x.add_row([1, 'KNeighbors Classifier', np.round(train_loss_knn, 4), np.round(test_loss_
x.add_row([2, 'Logistic Regression', np.round(train_loss_logistic, 4), np.round(test_lo
x.add_row([3, 'Support Vectors', np.round(train_loss_svc, 4), np.round(test_loss_svc, 4
x.add_row([4, 'Random Forest Classifier', np.round(train_loss_rf, 4), np.round(test_los
x.add_row([5, 'Light GBM Classifier', np.round(train_loss_lgbm, 4), np.round(test_loss_
x.add_row([6, 'Adaboost Classifier', np.round(train_loss_adaboost, 4), np.round(test_lo
x.add_row([7, 'MLP Classifier', np.round(train_loss_mlp, 4), np.round(test_loss_mlp, 4)
x.add_row([8, 'Stacking Classifier', np.round(train_loss_stacking, 4), np.round(test_lo

print(x)
```

```
+---------+--------------------------+----------------+---------------+----------------+
| Sr. No. |        Classifier        | Train Log-Loss | Test Log-Loss | Run-time (sec) |
+---------+--------------------------+----------------+---------------+----------------+
|    1    |  KNeighbors Classifier   |     0.0049     |     0.0286    |     0.001      |
|    2    |   Logistic Regression    |     0.3741     |     0.3734    |     0.0001     |
|    3    |     Support Vectors      |     0.0884     |     0.0883    |     0.0005     |
|    4    | Random Forest Classifier |     0.0041     |     0.0092    |     0.4032     |
|    5    |   Light GBM Classifier   |     0.0034     |     0.0083    |     0.0009     |
|    6    |   Adaboost Classifier    |     0.1077     |     0.108     |     0.0054     |
|    7    |      MLP Classifier      |     0.0102     |     0.0112    |     0.0361     |
|    8    |   Stacking Classifier    |     0.005      |     0.0108    |     0.0631     |
+---------+--------------------------+----------------+---------------+----------------+
```

From the results it is clear that **Light GBM Classifier** is the best model with test log-loss of 0.0083 and run-time of only 0.0009 seconds. Hence, the model satisfies the low latency requirement alongwith being capable of producing reliable predictions on firewall traffic logs.

6. Model Deployment:

The model can now be deployed in production by creating scripts with suitable data processing pipelines. We will create two scripts for deployment. The first script `Model.py` will train the best model on the data and second script `App.py` will implement flask api for deployment.

6.1. Model.py:

In [ ]:

```
import pandas as pd
import numpy as np
import joblib
```

```
from lightgbm import LGBMClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, RobustScaler
import networkx as nx

#####################################################

# reading dataframe
data = pd.read_csv("log2.csv")
# removing instances where Bytes are beyond 99th percentile
data = data[data['Bytes'] <= np.percentile(data['Bytes'], 99)]
# removing instances where Packets are beyond 99th percentile
data = data[data['Packets'] <= np.percentile(data['Packets'], 99)]
# removing instances where Elapsed Time (sec) are beyond 99th percentile
data = data[data['Elapsed Time (sec)'] <= np.percentile(data['Elapsed Time (sec)'], 99)
# adding translation features
data['Source Port Translation'] = (data['Source Port'] != data['NAT Source Port']).asty
data['Destination Port Translation'] = (data['Destination Port'] != data['NAT Destinati

# building bidirectional graph by using port numbers on host devices
HOST_NW = nx.DiGraph(name = "Host")
HOST_NW.add_edges_from(data[['Source Port', 'Destination Port']].values)
joblib.dump(HOST_NW, 'host_nw.pkl')
# building bidirectional graph by using port numbers on NAT
NAT_NW = nx.DiGraph(name = "NAT")
NAT_NW.add_edges_from(data[['NAT Source Port', 'NAT Destination Port']].values)
joblib.dump(NAT_NW, 'nat_nw.pkl')

def common_ports(nw, src, dst):
    """
    Counts no. of common ports connected directly between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        No. of common ports from intersection of set of neighbors of both src and dst p
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst))))
    except:
        return 0

# adding features to the dataset
data['Host CP'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT CP'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro

def jaccard_index(nw, src, dst):
    """
    Counts Jaccard index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Jaccard Index
```

```
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / len(s
    except:
        return 0

# adding features to the dataset
data['Host JI'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT JI'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro

def salton_index(nw, src, dst):
    """
    Counts Salton index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Salton Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / np.sq
    except:
        return 0

# adding features to the dataset
data['Host SL'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT SL'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro

def sorensen_index(nw, src, dst):
    """
    Counts Sorensen index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Sorensen Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / (len(
    except:
        return 0

# adding features to the dataset
data['Host SI'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT SI'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro

def adamic_adar_index(nw, src, dst):
    """
    Counts Adamic-Adar index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port
```

```
    Returns:
        Adamic-Adar Index
    """
    try:
        ports = set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))
        return 1/np.sum([np.log10(set(nw.neighbors(port))) for port in ports])
    except:
        return 0

# adding features to the dataset
data['Host AA'] = data.apply(lambda row: common_ports(HOST_NW, row['Source Port'], row[
data['NAT AA'] = data.apply(lambda row: common_ports(NAT_NW, row['NAT Source Port'], ro

# networkx library provides a function to calculate page rank of ports in the networks.
host_page_rank = nx.pagerank(HOST_NW)
nat_page_rank = nx.pagerank(NAT_NW)

# adding features to the dataset
data['Host Source PR'] = data.apply(lambda row: host_page_rank.get(row['Source Port'],
data['Host Destination PR'] = data.apply(lambda row: host_page_rank.get(row['Destinatio
data['NAT Source PR'] = data.apply(lambda row: nat_page_rank.get(row['NAT Source Port']
data['NAT Destination PR'] = data.apply(lambda row: nat_page_rank.get(row['NAT Destinat

# seperating Action class as target feature
y = data['Action']
X = data.drop(['Action'], axis = 1)

# splitting data for calibration
X_train, X_cv, y_train, y_cv = train_test_split(X, y, test_size = 0.20, stratify = y, r

# applying RobustScaler
robust_scaler = RobustScaler()
robust_scaler_features = ['Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed
X_train_robust_scaled = robust_scaler.fit_transform(X_train[robust_scaler_features])
X_cv_robust_scaled = robust_scaler.transform(X_cv[robust_scaler_features])
joblib.dump(robust_scaler, 'robust_scaler.pkl')

# applying StandardScaler
std_scaler = StandardScaler()
std_scaler_features = ['Host CP', 'NAT CP', 'Host JI', 'NAT JI', 'Host SL', 'NAT SL', '
X_train_std_scaled = std_scaler.fit_transform(X_train[std_scaler_features])
X_cv_std_scaled = std_scaler.transform(X_cv[std_scaler_features])
joblib.dump(std_scaler, 'std_scaler.pkl')

# stacking the scaled features
X_train_preprocessed = np.hstack((X_train[X_train.columns[:4]], X_train_robust_scaled,
X_cv_preprocessed = np.hstack((X_cv[X_cv.columns[:4]], X_cv_robust_scaled, X_cv_std_sca

# training the classifier on data
lgbm = LGBMClassifier(n_estimators = 750, max_depth = 4, objective = 'multiclass', clas
lgbm.fit(X_train_preprocessed, y_train)

# calibrating the classifier on validation data
calibrator_lgbm = CalibratedClassifierCV(lgbm, method = 'isotonic', cv = 'prefit')
calibrator_lgbm.fit(X_cv_preprocessed, y_cv)
joblib.dump(calibrator_lgbm, 'calibrator_lgbm.pkl')
```

6.2. App.py:

In [ ]:

```
from flask import Flask, jsonify, request
import numpy as np
import pandas as pd
from bs4 import BeautifulSoup
import re
import joblib
import networkx as nx
from lightgbm import LGBMClassifier

# loading pickle objects
robust_scaler = joblib.load('robust_scaler.pkl')
robust_scaler_features = ['Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed

std_scaler = joblib.load('std_scaler.pkl')
std_scaler_features = ['Host CP', 'NAT CP', 'Host JI', 'NAT JI', 'Host SL', 'NAT SL', '

calibrator_lgbm = joblib.load('calibrator_lgbm.pkl')
HOST_NW = joblib.load('host_nw.pkl')
NAT_NW = joblib.load('nat_nw.pkl')

# https://www.tutorialspoint.com/flask
import flask
app = Flask(__name__)


#####################################################


def common_ports(nw, src, dst):
    """
    Counts no. of common ports connected directly between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        No. of common ports from intersection of set of neighbors of both src and dst p
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst))))
    except:
        return 0


def jaccard_index(nw, src, dst):
    """
    Counts Jaccard index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Jaccard Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / len(s
    except:
```

```
        return 0


def salton_index(nw, src, dst):
    """
    Counts Salton index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Salton Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / np.sq
    except:
        return 0

def sorensen_index(nw, src, dst):
    """
    Counts Sorensen index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Sorensen Index
    """
    try:
        return len(set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))) / (len(
    except:
        return 0

def adamic_adar_index(nw, src, dst):
    """
    Counts Adamic-Adar index between src and dst ports.

    Args:
        nw: Network instance
        src: Source Port
        dst: Destination Port

    Returns:
        Adamic-Adar Index
    """
    try:
        ports = set(nw.neighbors(src)).intersection(set(nw.neighbors(dst)))
        return 1/np.sum([np.log10(set(nw.neighbors(port))) for port in ports])
    except:
        return 0

# networkx library provides a function to calculate page rank of ports in the networks.
host_page_rank = nx.pagerank(HOST_NW)
nat_page_rank = nx.pagerank(NAT_NW)


def preprocess(feature_vector):
```

```
    """
    Preprocesses the feature matrix of firewall logs.

    Args:
        feature_vector: Input feature matrix of firewall logs

    Returns:
        Preprocessed feature vector
    """
    try:
        # reshaping into row vector if feature matrix of single instance is given
        # creating empty dataframe
        feature_names = ['Source Port', 'Destination Port', 'NAT Source Port', 'NAT Des
                         'Bytes Received', 'Packets', 'Elapsed Time (sec)', 'pkts_sent'
        data_matrix = pd.DataFrame(feature_vector, columns = feature_names)
        # applying engineered features
        data_matrix['Source Port Translation'] = (data_matrix['Source Port'] != data_ma
        data_matrix['Destination Port Translation'] = (data_matrix['Destination Port']
        data_matrix['Host CP'] = data_matrix.apply(lambda row: common_ports(HOST_NW, ro
        data_matrix['NAT CP'] = data_matrix.apply(lambda row: common_ports(NAT_NW, row[
        data_matrix['Host JI'] = data_matrix.apply(lambda row: jaccard_index(HOST_NW, r
        data_matrix['NAT JI'] = data_matrix.apply(lambda row: jaccard_index(NAT_NW, row
        data_matrix['Host SL'] = data_matrix.apply(lambda row: salton_index(HOST_NW, ro
        data_matrix['NAT SL'] = data_matrix.apply(lambda row: salton_index(NAT_NW, row[
        data_matrix['Host SI'] = data_matrix.apply(lambda row: sorensen_index(HOST_NW,
        data_matrix['NAT SI'] = data_matrix.apply(lambda row: sorensen_index(NAT_NW, ro
        data_matrix['Host AA'] = data_matrix.apply(lambda row: adamic_adar_index(HOST_N
        data_matrix['NAT AA'] = data_matrix.apply(lambda row: adamic_adar_index(NAT_NW,
        data_matrix['Host Source PR'] = data_matrix.apply(lambda row: host_page_rank.ge
        data_matrix['Host Destination PR'] = data_matrix.apply(lambda row: host_page_ra
        data_matrix['NAT Source PR'] = data_matrix.apply(lambda row: nat_page_rank.get(
        data_matrix['NAT Destination PR'] = data_matrix.apply(lambda row: nat_page_rank
        # scaling the data
        data_matrix_robust_scaled = robust_scaler.transform(data_matrix[robust_scaler_f
        data_matrix_std_scaled = std_scaler.transform(data_matrix[std_scaler_features])
        data_matrix_preprocessed = np.hstack((data_matrix[data_matrix.columns[:4]], dat
        return data_matrix_preprocessed
    except:
        print("The last dimension of the data should be 11")



@app.route('/')
def hello_world():
    return 'Hello World!'


@app.route('/index')
def index():
    return flask.render_template('index.html')


@app.route('/predict', methods=['POST'])
def predict():
    """
    Predicts the firewall Action depending on the feature_matrix of firewall logs

    Returns:
        Predicted Action class
    """
    inp = request.form.to_dict()
```

```
    feature_vector = np.array([inp['source_p'], inp['dest_p'], inp['nat_source_p'], inp
                    inp['bts_sent'], inp['bts_received'], inp['pkts'], inp['elapsed_t']
    data_preprocessed = preprocess(feature_vector.reshape(1, -1))
    prediction = calibrator_lgbm.predict(data_preprocessed.reshape(1, -1))
    return jsonify({'Predicted Action': str(prediction)})


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

The model can now be deployed by using flask API on cloud platforms such as Amazon AWS and GCP.