CNN-LSTM: Network Intrusion Detection System

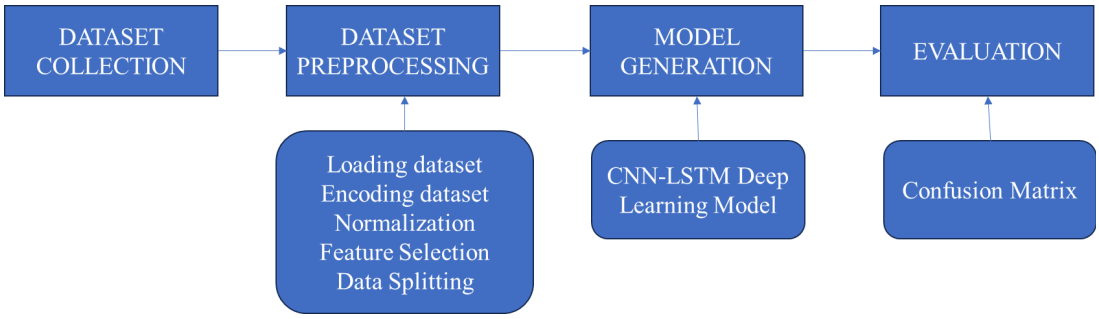
1.Overview

The CNN-LSTM model combines the strengths of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks to effectively handle data with spatial and temporal dependencies. CNNs are well-suited for feature extraction from structured data like images, while LSTM networks are designed to capture temporal relationships in sequential data.

Dataset:

The UNSW-NB15 dataset is well known for its detailed coverage of contemporary network traffic, encompassing both legitimate and harmful actions. It includes nine different forms of attacks including DoS, Exploits, Fuzzers, Backdoor, and Analysis. The data was gathered from actual traffic sources, making it very suitable for assessing the IDS effectiveness.

2. Model Architecture:



The architecture of the CNN-LSTM model used in this project consists of:

- **Convolutional Layers:** These layers perform feature extraction by applying convolutional filters to the input data, capturing spatial features and patterns.
- **Pooling Layers:** Max-pooling or average-pooling layers reduce the dimensionality of the data, making the model more computationally efficient.
- **LSTM Layers:** The extracted features from the CNN are fed into LSTM layers, which process the sequential dependencies and learn temporal patterns.
- **Dense Layers:** After the LSTM layers, dense (fully connected) layers are used to make predictions.

3. Training and Optimization

The model is trained using a dataset where features are fed into the CNN layers followed by LSTM for sequence processing. Techniques like early stopping and learning rate scheduling are used to avoid overfitting and optimize the model training process.

4. RESULTS & DISCUSSION

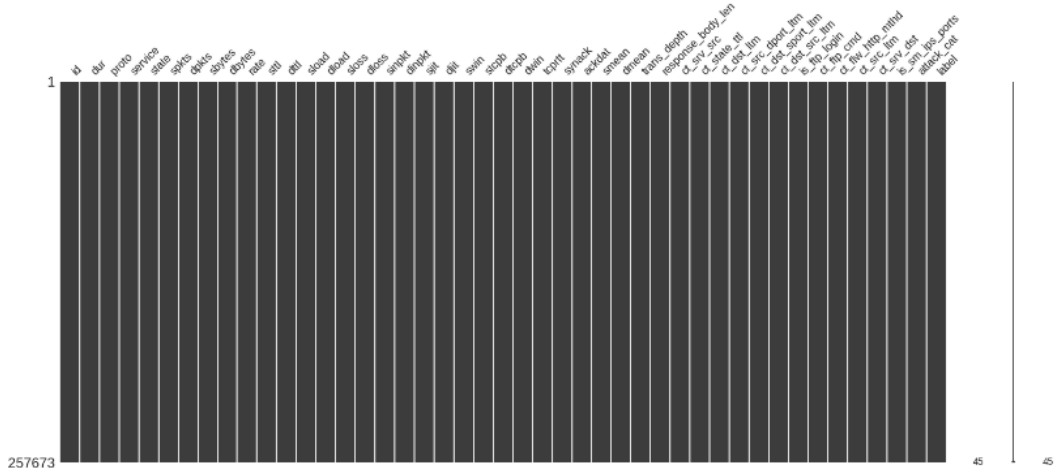
LOADING DATASET ON KAGGLE:

UNSW-NB15 dataset is a benchmark dataset for evaluating network intrusion detection systems. It contains network traffic data with labeled instances of various attack types and normal activities, captured in four separate CSV files. Additionally, a features file provides detailed descriptions of each attribute. For analysis, these data files are combined into a single dataset to ensure consistency during model training and testing.

```
/kaggle/input/unsw-nb15/UNSW-NB15_2.csv
/kaggle/input/unsw-nb15/NUSW-NB15_features.csv
/kaggle/input/unsw-nb15/UNSW-NB15_1.csv
/kaggle/input/unsw-nb15/UNSW_NB15_testing-set.csv
/kaggle/input/unsw-nb15/UNSW-NB15_3.csv
/kaggle/input/unsw-nb15/UNSW_NB15_training-set.csv
/kaggle/input/unsw-nb15/UNSW-NB15_LIST_EVENTS.csv
/kaggle/input/unsw-nb15/UNSW-NB15_4.csv
```

MISSING VALUES:

There are no missing values in the UNSW-NB15 dataset, ensuring that the data is complete and does not require imputation or additional preprocessing for handling missing entries. This allows for direct application of machine learning models without concerns about data gaps impacting the analysis.



Data is clean and there are no missing values.

PREPROCESSING:

During preprocessing, the UNSW-NB15 dataset was first loaded and combined from its four CSV files into a single cohesive dataset. Basic exploratory data analysis included printing the first few rows of the dataset to understand its structure, inspecting the data types, and verifying the absence of missing values, which facilitated a smooth workflow for further processing steps such as scaling and encoding.

Name	srcip	sport	dstip	dsport	proto	state	dur	sbytes	dbytes	sttl	...	ct_ftp_cmd	ct_srv_src	ct_srv_dst	ct_dst_ltm	ct_src_ltm	ct_src_dport_ltm	ct_dst_spor
0	59.166.0.0	1390	149.171.126.6	53	udp	CON	0.001055	132	164	31	...	0	3	7	1	3	1	
1	59.166.0.0	33661	149.171.126.9	1024	udp	CON	0.036133	528	304	31	...	0	2	4	2	3	1	
2	59.166.0.6	1464	149.171.126.7	53	udp	CON	0.001119	146	178	31	...	0	12	8	1	2	2	
3	59.166.0.5	3593	149.171.126.5	53	udp	CON	0.001209	132	164	31	...	0	6	9	1	1	1	
4	59.166.0.3	49664	149.171.126.0	53	udp	CON	0.001169	146	178	31	...	0	7	9	1	1	1	

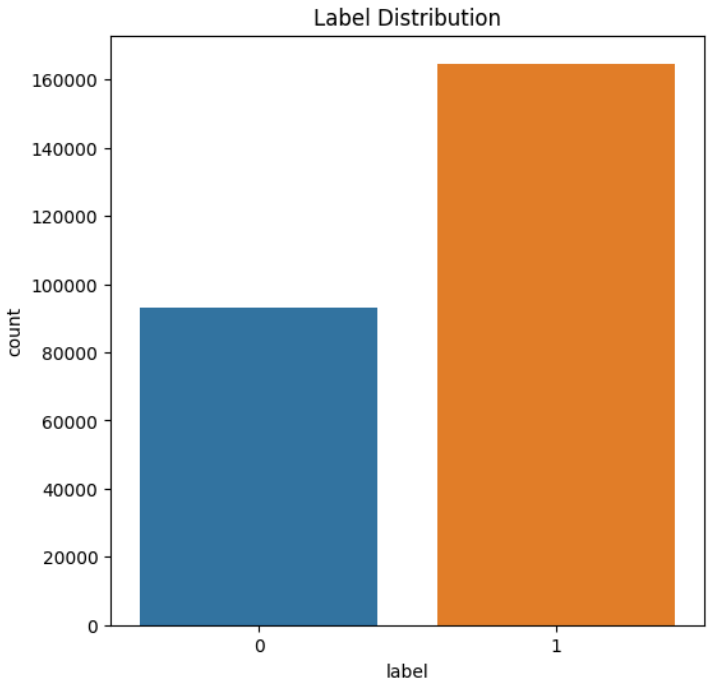
5 rows × 49 columns

In the preprocessing stage, specific columns in the dataset were addressed to ensure data consistency and usability. The `ct_ftp_cmd` column, which contained some blank values, was treated by replacing empty entries with 0 and converting the data type to integers. Additionally, key columns such as `service`, `ct_flw_http_mthd`, `is_ftp_login`, `ct_ftp_cmd`, `attack_cat`, and `Label` were selected for further analysis.

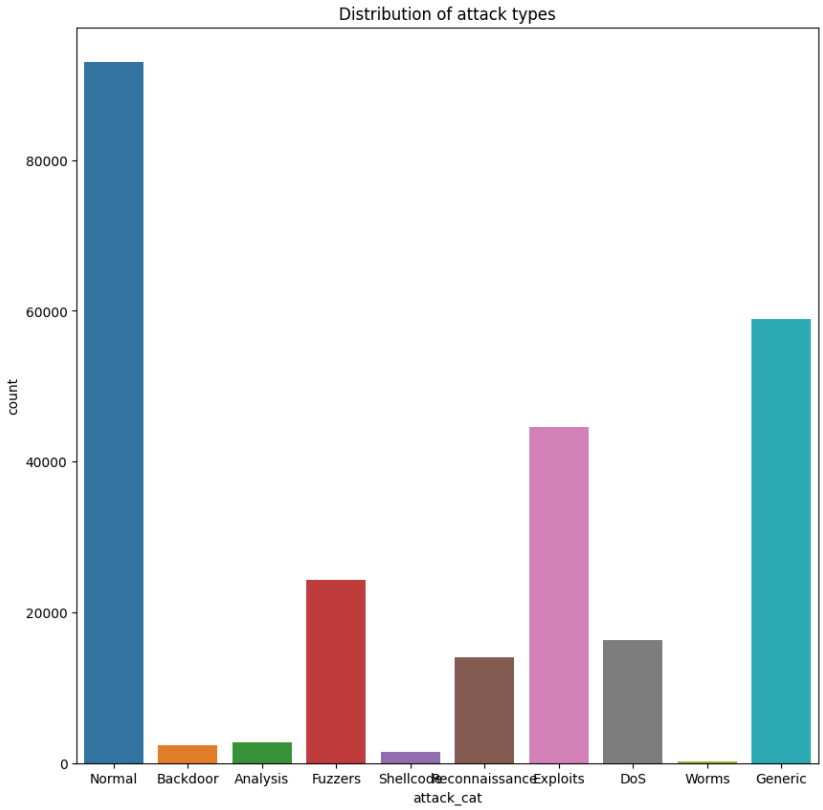
Name	service	ct_flw_http_mthd	is_ftp_login	ct_ftp_cmd	attack_cat	Label
0	dns	0.0	0.0	0	normal	0
1	None	0.0	0.0	0	normal	0
2	dns	0.0	0.0	0	normal	0
3	dns	0.0	0.0	0	normal	0
4	dns	0.0	0.0	0	normal	0
...	...	...	...	...	...	...
2540042	ftp-data	0.0	0.0	0	normal	0
2540043	ftp	0.0	1.0	2	normal	0
2540044	ftp	0.0	1.0	2	normal	0
2540045	http	2.0	0.0	0	normal	0
2540046	pop3	0.0	0.0	0	exploits	1

2540047 rows × 6 columns

To understand the dataset's composition, the distribution of labels and attack categories was examined. The `Label` column, which indicates whether an instance is benign or an attack, was analyzed to identify any class imbalance.



The `attack_cat` column, representing various types of attacks, was evaluated to determine the frequency of each attack type. This analysis provided insights into the prevalence of different attack categories, which is essential for guiding the model's training process and addressing any potential biases in classification.

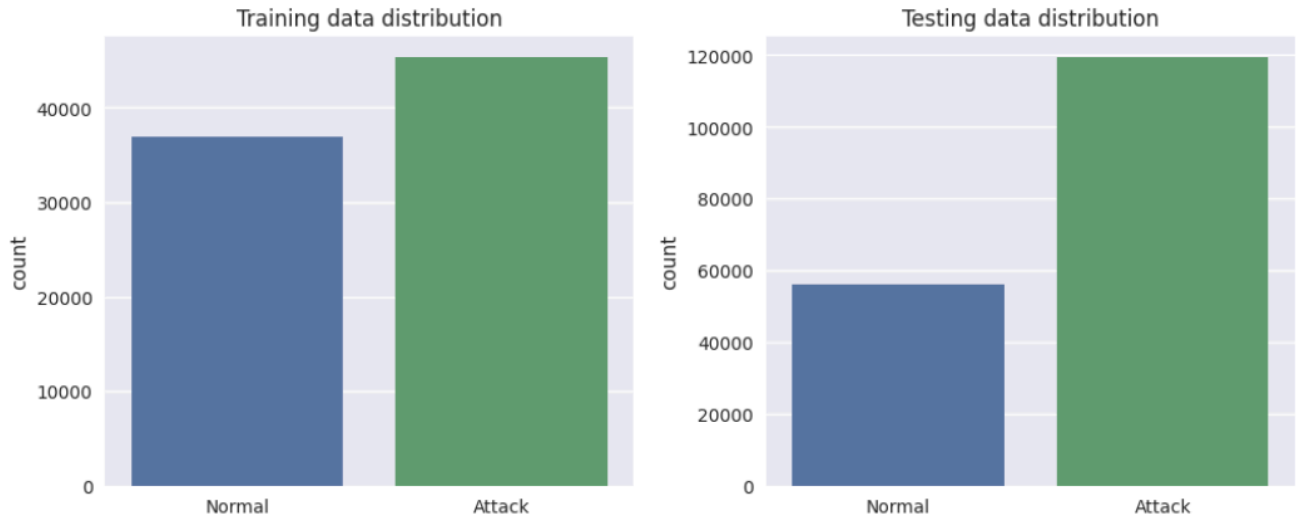


```
attack_cat
Generic             0.670689
Exploits            0.138585
 Fuzzers            0.059745
DoS                 0.050899
 Reconnaissance     0.038060
 Fuzzers            0.015721
Analysis            0.008332
Backdoor            0.005587
Reconnaissance      0.005475
 Shellcode          0.004009
Backdoors           0.001662
Shellcode           0.000694
Worms               0.000542
Name: proportion, dtype: float64
```

TRAINING AND TESTING DATASET:

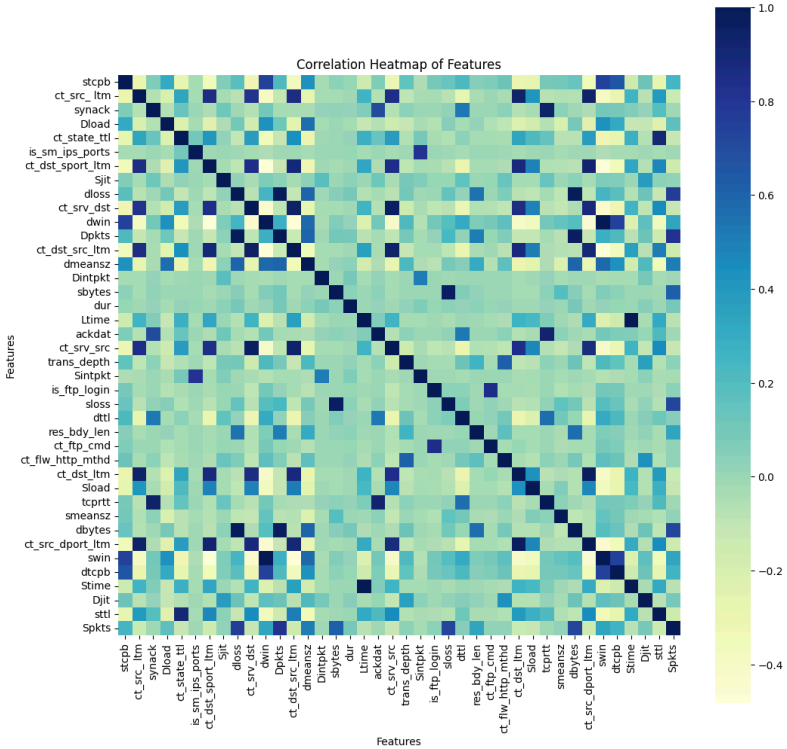
The training and testing datasets length and distribution were analyzed to understand the distribution of instances across different attack categories.

```
Train data length: 82332
Test data length: 175341
```



CORRELATION HEATMAP OF FEATURES:

A correlation heatmap was generated to visualize the relationships between various features in the dataset. This graphical representation highlights how strongly features are correlated with one another, with values ranging from -1 to 1, where -1 indicates a perfect negative correlation, 0 indicates no correlation, and 1 indicates a perfect positive correlation. The heatmap revealed interesting patterns, such as strong correlations between certain numerical features.



TRAIN TEST SPLIT:

The dataset was split into training and testing subsets using an 80-20 split ratio. This approach allocates 80% of the data for training the model and reserves 20% for testing its performance. Additionally, the training data underwent further division into training and validation sets, maintaining an 80-20 split within the training portion.

```
train, test = train_test_split(combined_data,test_size=0.2,random_state=16)
train, val = train_test_split(train,test_size=0.2,random_state=16)
train.shape
```

```
(1625629, 44)
```

```
test.shape
```

```
(508010, 44)
```

```
x_train, y_train = train.drop(columns=['attack_cat']), train[['attack_cat']]
x_test, y_test = test.drop(columns=['attack_cat']), test[['attack_cat']]
x_val, y_val = val.drop(columns=['attack_cat']), val[['attack_cat']]
x_train.shape, y_train.shape
```

```
((1625629, 43), (1625629, 1))
```

```
x_test.shape, y_test.shape
```

```
((508010, 43), (508010, 1))
```

```
x_val.shape, y_val.shape
```

```
((406408, 43), (406408, 1))
```

```
y_train.columns
```

```
Index(['attack_cat'], dtype='object', name='Name')
```

```
attacks = y_train['attack_cat'].unique()
attacks
```

```
array(['normal', 'generic', 'fuzzers', 'exploits', 'dos',
       'reconnaissance', 'backdoor', 'analysis', 'shellcode', 'worms'],
      dtype=object)
```

```
y_train.shape
```

```
(1625629, 10)
```

```
x_train.shape
```

```
(1625629, 204)
```

```
x_test.shape
```

```
(508010, 204)
```

TRAIN DATA:

The training dataset comprises 64% of the total dataset, containing a balanced representation of both benign and attack instances. Analyzing the training data revealed that certain attack types were more frequent, providing the model with ample examples to learn from while also allowing it to understand the nuances of less common attacks.

Name	proto	state	dur	sbytes	dbytes	sttl	dttl	sloss	dloss	service	...	is_ftp_login	ct_ftp_cmd	ct_srv_src	ct_srv_dst	ct_dst_ltm	ct_src_ltm	ct_src_dport_ltm	ct_dst_sp
1070804	tcp	FIN	0.438857	4776	3080	31	29	7	7	None	...	0.0	0	6	3	4	5	1	
1054569	tcp	CON	0.026163	2230	13900	31	29	7	10	None	...	0.0	0	2	7	2	1	1	
1548299	tcp	FIN	0.014954	2854	26584	31	29	7	16	None	...	0.0	0	9	6	4	8	1	
2305760	udp	INT	0.000009	114	0	254	0	0	0	dns	...	0.0	0	33	33	17	17	17	
1989356	udp	CON	0.001562	544	304	31	29	0	0	None	...	0.0	0	6	7	5	6	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
358801	tcp	FIN	0.046476	320	1890	31	29	1	2	ftp-data	...	0.0	0	3	5	3	7	1	
556159	tcp	FIN	0.027755	5928	8010	31	29	14	17	ssh	...	0.0	0	1	1	8	3	1	
1408939	udp	INT	0.000003	264	0	60	0	0	0	dns	...	0.0	0	39	39	18	18	18	
1894282	tcp	FIN	0.057911	2766	27392	31	29	7	16	None	...	0.0	0	2	2	3	4	1	
2148776	tcp	FIN	0.263385	1920	4416	31	29	6	6	None	...	0.0	0	9	4	6	9	2	

1625629 rows × 44 columns

TEST DATA:

The testing dataset represents 20% of the total dataset and serves as an independent benchmark for evaluating the model's performance. By retaining a similar attack-wise distribution, the test data allows for a fair comparison of model predictions against actual labels.

Name	proto	state	dur	sbytes	dbytes	sttl	dttl	sloss	dloss	service	...	is_ftp_login	ct_ftp_cmd	ct_srv_src	ct_srv_dst	ct_dst_ltm	ct_src_ltm	ct_src_dport_ltm	ct_dst_sp
1070804	tcp	FIN	0.438857	4776	3080	31	29	7	7	None	...	0.0	0	6	3	4	5	1	
1054569	tcp	CON	0.026163	2230	13900	31	29	7	10	None	...	0.0	0	2	7	2	1	1	
1548299	tcp	FIN	0.014954	2854	26584	31	29	7	16	None	...	0.0	0	9	6	4	8	1	
2305760	udp	INT	0.000009	114	0	254	0	0	0	dns	...	0.0	0	33	33	17	17	17	
1989356	udp	CON	0.001562	544	304	31	29	0	0	None	...	0.0	0	6	7	5	6	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
358801	tcp	FIN	0.046476	320	1890	31	29	1	2	ftp-data	...	0.0	0	3	5	3	7	1	
556159	tcp	FIN	0.027755	5928	8010	31	29	14	17	ssh	...	0.0	0	1	1	8	3	1	
1408939	udp	INT	0.000003	264	0	60	0	0	0	dns	...	0.0	0	39	39	18	18	18	
1894282	tcp	FIN	0.057911	2766	27392	31	29	7	16	None	...	0.0	0	2	2	3	4	1	
2148776	tcp	FIN	0.263385	1920	4416	31	29	6	6	None	...	0.0	0	9	4	6	9	2	

1625629 rows × 44 columns

x_train and x_test:

```
print(x_train)
```

```
[[ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.34135356]
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 ...
 [ 0.          0.          0.         ...  1.5752582   2.33470341
   2.85751364]
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 [ 0.          0.          0.         ... -0.31142568 -0.25758182
  -0.43021098]]
```

+ Code + Markdown

```
print(x_test)
```

```
[[ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 [ 0.          0.          0.         ...  2.4006824   3.4688282
   2.14665427]
 ...
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.25249614]
 [ 0.          0.          0.         ... -0.42934342 -0.41959965
  -0.51906841]
 [ 0.          0.          0.         ...  1.45734046  2.17268558
   0.90265035]]
```

UNIQUE COUNTS:

```
unique_values, counts = np.unique(y_train, return_counts=True)
for value, count in zip(unique_values, counts):
    print(f"Value: {value}, Count: {count}")
```

```
Value: analysis, Count: 1716
Value: backdoor, Count: 1499
Value: dos, Count: 10454
Value: exploits, Count: 28640
Value: fuzzers, Count: 15494
Value: generic, Count: 137574
Value: normal, Count: 1420187
Value: reconnaissance, Count: 8985
Value: shellcode, Count: 979
Value: worms, Count: 101
```

```
unique_values, counts = np.unique(y_test, return_counts=True)


for value, count in zip(unique_values, counts):
    print(f"Value: {value}, Count: {count}")
```

```
Value: analysis, Count: 544
Value: backdoor, Count: 470
Value: dos, Count: 3288
Value: exploits, Count: 8709
Value: fuzzers, Count: 4928
Value: generic, Count: 43187
Value: normal, Count: 443714
Value: reconnaissance, Count: 2813
Value: shellcode, Count: 320
Value: worms, Count: 37
```

TRAINING MODEL:

The model was designed using a hybrid architecture that combines Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks to effectively analyze sequential data, such as network traffic patterns. The model is defined with a series of convolutional and LSTM layers, allowing it to capture both local features and long-term dependencies in the data. The training process involved fitting this model to the preprocessed training data, enabling it to learn to distinguish between different classes of network activities.

The architecture of the model was constructed using the Sequential API from Keras, and it includes multiple convolutional blocks followed by LSTM layers. Each layer transforms the input data into progressively more abstract representations, ultimately producing predictions for the target classes. Below is a summary table detailing the model architecture, including the type of layers, their output shapes, and the number of parameters:

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv1d_6 (Conv1D)	(None, 204, 16)	32
max_pooling1d_6 (MaxPooling1D)	(None, 102, 16)	0
batch_normalization_6 (BatchNormalization)	(None, 102, 16)	64
lstm_6 (LSTM)	(None, 102, 16)	2,112
conv1d_7 (Conv1D)	(None, 100, 32)	1,568
max_pooling1d_7 (MaxPooling1D)	(None, 50, 32)	0
batch_normalization_7 (BatchNormalization)	(None, 50, 32)	128
lstm_7 (LSTM)	(None, 50, 32)	8,320
conv1d_8 (Conv1D)	(None, 46, 64)	10,304
max_pooling1d_8 (MaxPooling1D)	(None, 23, 64)	0
batch_normalization_8 (BatchNormalization)	(None, 23, 64)	256
lstm_8 (LSTM)	(None, 64)	33,024
dense_4 (Dense)	(None, 64)	4,160
dropout_2 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 10)	650

Total params: 60,618 (236.79 KB)

Trainable params: 60,394 (235.91 KB)

Non-trainable params: 224 (896.00 B)

The model was compiled using the Adam optimizer and categorical cross entropy as the loss function, suitable for multi-class classification problems. In addition, precision and recall metrics were included to evaluate the model's performance on imbalanced datasets. The model's architecture and configuration are optimized for learning intricate patterns in the data, providing a solid foundation for accurate classification of network activities.

Model Performance Metrics

The performance of the model was evaluated using several key metrics, including accuracy, precision, and recall, on the test dataset. The test accuracy achieved was 97.59%, indicating that the model correctly classified approximately 97.59% of the instances in the test set. The test precision was reported at 99.08%, demonstrating that when the model predicted an attack, it was accurate 99.08% of the time. The test recall was 96.60%, meaning the model successfully identified 96.60% of all actual attack instances present in the dataset. These metrics highlight the model's strong ability to differentiate between normal and attack traffic.

```
history = model.fit(x_train, y_train, epochs=2, batch_size=512,validation_data=(x_val,y_val))

 #Evaluate the model
test_loss, test_accuracy, test_precision, test_recall = model.evaluate(x_test, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
print("Test Precision:", test_precision)
print("Test Recall:", test_recall)
```

```
Epoch 1/2
3176/3176 ━━━━━━━━━━━━━━━━━━━━ 1010s 315ms/step - accuracy: 0.9647 - loss: 0.1082 - precision_3: 0.9871 - recall_3: 0.9457 - val_accuracy:
0.9757 - val_loss: 0.0620 - val_precision_3: 0.9892 - val_recall_3: 0.9664
Epoch 2/2
3176/3176 ━━━━━━━━━━━━━━━━━━━━ 990s 312ms/step - accuracy: 0.9756 - loss: 0.0623 - precision_3: 0.9896 - recall_3: 0.9657 - val_accuracy:
0.9761 - val_loss: 0.0597 - val_precision_3: 0.9909 - val_recall_3: 0.9660
15876/15876 ━━━━━━━━━━━━━━━━━━━━ 413s 26ms/step - accuracy: 0.9757 - loss: 0.0603 - precision_3: 0.9908 - recall_3: 0.9655
Test Loss: 0.05975496396422386
Test Accuracy: 0.9759394526481628
Test Precision: 0.9908214807510376
Test Recall: 0.9660046100616455
```

- **Test Accuracy:** 97.59 %
- **Test Precision:** 99.08 %
- **Test Recall:** 96.60 %

ACCURACY: Average K-Fold Cross-Validation Results (on Validation Set):

The model's robustness was further assessed through K-Fold Cross-Validation, yielding average results on the validation set. The average accuracy was 97.58%, which closely mirrors the test accuracy, reinforcing the model's reliability. The average precision during cross-validation was 99.00%, and the average recall was 96.58%. These results confirm the model's effectiveness in maintaining high performance across different subsets of data, ensuring it generalizes well to unseen instances.

```
y_train
```

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

```
from sklearn.model_selection import StratifiedKFold
kfold = StratifiedKFold(n_splits=2, shuffle=True, random_state=42)
y_train_labels = np.argmax(y_train, axis=1)
y_train_labels
```

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

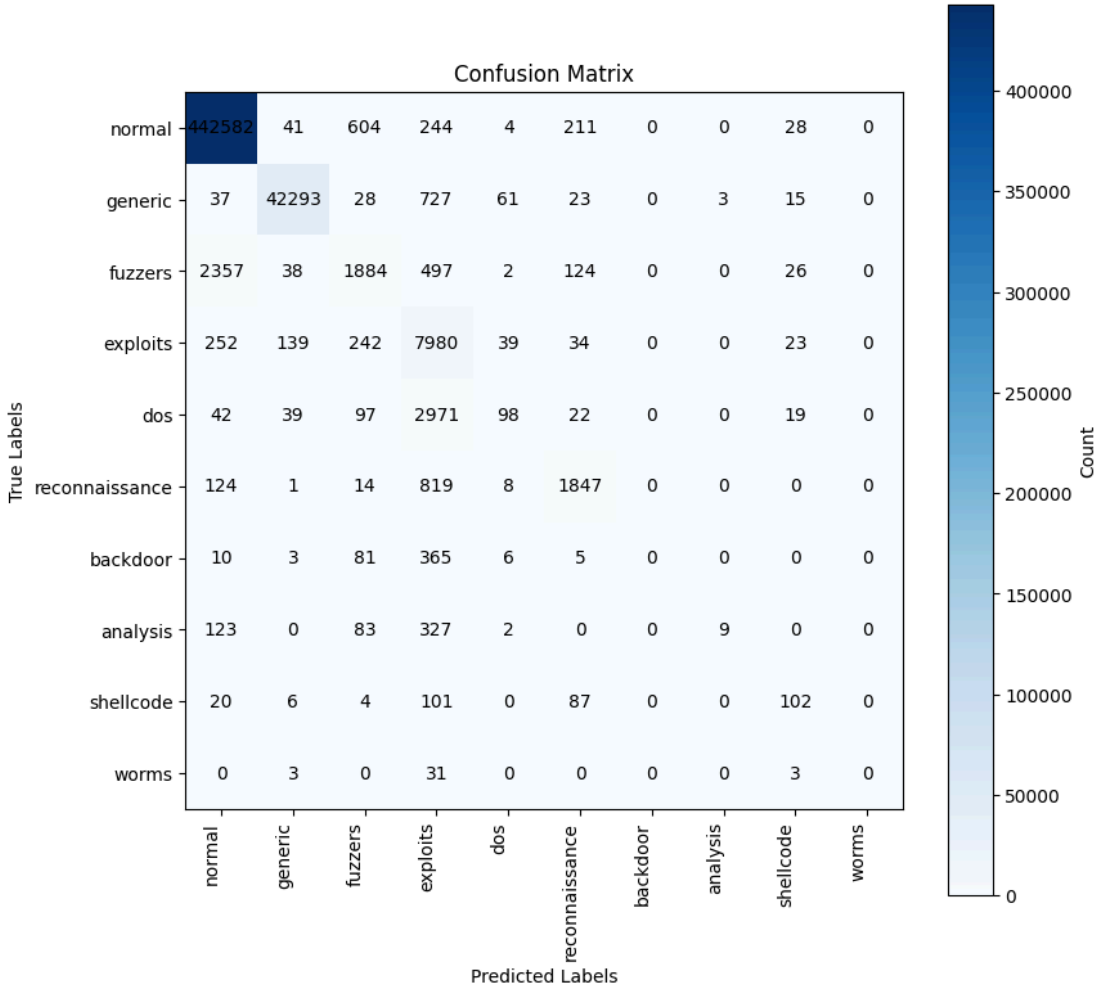
```
scores = []
model = create_model()
for train_index, val_index in kfold.split(x_train, y_train_labels):
    X_train_inner, X_val_inner = x_train[train_index], x_train[val_index]
    y_train_inner, y_val_inner = y_train[train_index], y_train[val_index]

    model.fit(X_train_inner, y_train_inner, epochs=2, batch_size=1024,validation_data=(X_val_inner, y_val_inner))
    test_loss, test_acc, precision, recall = model.evaluate(x_val, y_val)
    scores.append([test_loss, test_acc, precision, recall])

print("Average K-Fold Cross-Validation Results (on Validation Set):")
print("Loss:", np.mean([score[0] for score in scores]))
print("Accuracy:", np.mean([score[1] for score in scores]))
print("Precision:", np.mean([score[2] for score in scores]))
print("Recall:", np.mean([score[3] for score in scores]))
```

```
Epoch 1/2
794/794 ━━━━━━━━━━━━━━━━━━━━ 549s 680ms/step - accuracy: 0.9447 - loss: 0.1940 - precision_4: 0.9775 - recall_4: 0.9063 - val_accuracy: 0.9
716 - val_loss: 0.0721 - val_precision_4: 0.9909 - val_recall_4: 0.9589
Epoch 2/2
794/794 ━━━━━━━━━━━━━━━━━━━━ 547s 689ms/step - accuracy: 0.9733 - loss: 0.0689 - precision_4: 0.9901 - recall_4: 0.9616 - val_accuracy: 0.9
751 - val_loss: 0.0630 - val_precision_4: 0.9906 - val_recall_4: 0.9638
12701/12701 ━━━━━━━━━━━━━━━━━━━━ 317s 25ms/step - accuracy: 0.9756 - loss: 0.0620 - precision_4: 0.9910 - recall_4: 0.9646
Epoch 1/2
794/794 ━━━━━━━━━━━━━━━━━━━━ 536s 675ms/step - accuracy: 0.9747 - loss: 0.0639 - precision_4: 0.9895 - recall_4: 0.9641 - val_accuracy: 0.9
761 - val_loss: 0.0606 - val_precision_4: 0.9895 - val_recall_4: 0.9665
Epoch 2/2
794/794 ━━━━━━━━━━━━━━━━━━━━ 537s 677ms/step - accuracy: 0.9756 - loss: 0.0616 - precision_4: 0.9897 - recall_4: 0.9658 - val_accuracy: 0.9
765 - val_loss: 0.0590 - val_precision_4: 0.9894 - val_recall_4: 0.9674
12701/12701 ━━━━━━━━━━━━━━━━━━━━ 334s 26ms/step - accuracy: 0.9766 - loss: 0.0587 - precision_4: 0.9894 - recall_4: 0.9675
Average K-Fold Cross-Validation Results (on Validation Set):
Loss: 0.060666393488645554
Accuracy: 0.9758727550506592
Precision: 0.9900006651878357
Recall: 0.9658618867397308
```

- **Accuracy:** 97.58 %
- **Precision:** 99.00 %
- **Recall:** 96.58 %

Confusion Matrix

The confusion matrix provides additional insights into the model's classification performance, detailing how many instances were correctly and incorrectly classified for each class label.



Class Labels:

- Label 0: Represents normal network traffic, Label 1: Represents attack traffic.

```
unique_values, counts = np.unique(y_test, return_counts=True)

# Print the unique values and their corresponding counts
for value, count in zip(unique_values, counts):
    print(f"Value: {value}, Count: {count}")
```

```
Value: 0.0, Count: 4572090
Value: 1.0, Count: 508010
```

COMPARING WITH OTHER MODELS:

In this study, we compare the performance of our CNN-LSTM model against several traditional machine learning algorithms, including Logistic Regression, Support Vector Machine (SVM), and Gaussian Naive Bayes. By evaluating key performance metrics such as accuracy, precision, recall, F1-score, and Area Under the Curve (AUC) for each model, we aim to assess the strengths and weaknesses of the CNN-LSTM architecture in relation to these established methods. This comparative analysis will provide insights into the effectiveness of deep learning techniques for our specific task and highlight scenarios where CNN-LSTM may offer advantages or face limitations compared to more conventional approaches.

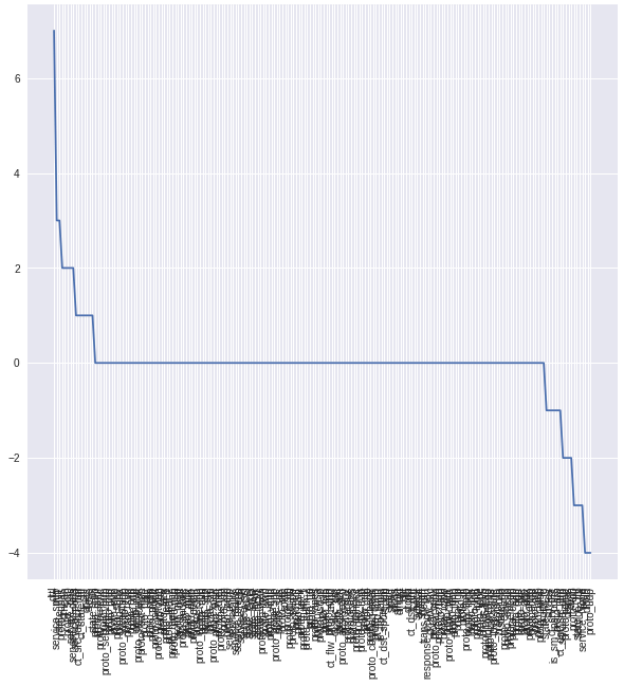
LOGISTIC REGRESSION:

[28]:

```
from sklearn.linear_model import LogisticRegression
start_time = time.time()
pred_now, acc_lr, acc_cv_lr, lr = fit_algo(LogisticRegression(C=0.1)
                                           , X, Y, 10)

lr_time = (time.time() - start_time)

print("Accuracy: %s" % acc_lr)
print("Accuracy of CV: %s" % acc_cv_lr)
print("Execution time: %s" % lr_time)
```

```
Accuracy: 90.1
Accuracy of CV: 89.05
Execution time: 61.02007722854614
```

FEATURE IMPORTANCE PLOT:



GAUSSIAN NAIVE BAYES:

```
from sklearn.naive_bayes import GaussianNB
start_time = time.time()

pred_now, acc_gnb, acc_cv_gnb, gnb= fit_algo(GaussianNB()
                                        ,X,Y,5)

gnb_time = (time.time() - start_time)

print("Accuracy: %s" % acc_gnb)
print("Accuracy of CV: %s" % acc_cv_gnb)
print("Execution time: %s" % gnb_time)
```

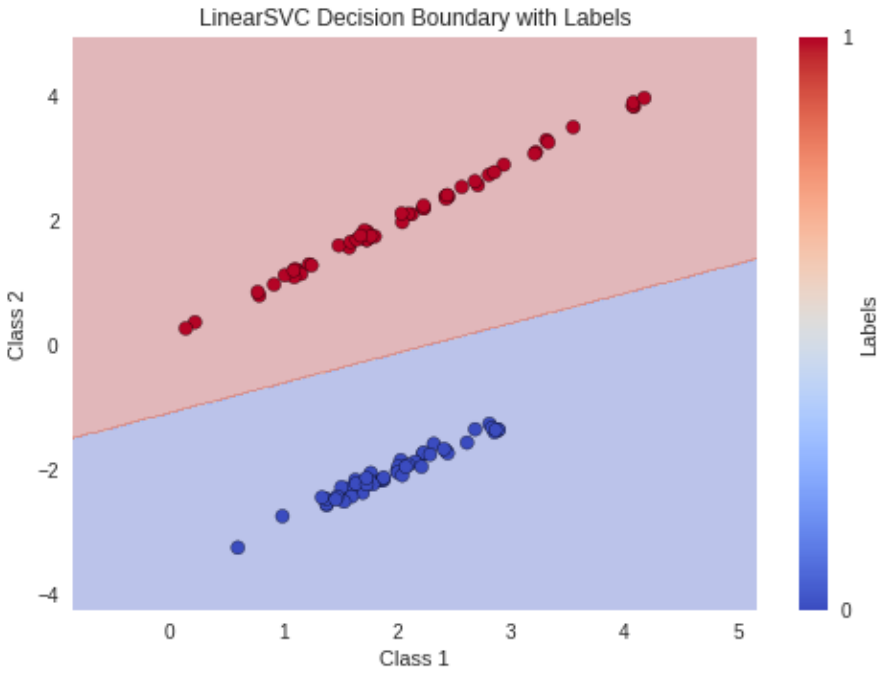
```
Accuracy: 50.5
Accuracy of CV: 50.46
Execution time: 9.620088338851929
```

SVM:

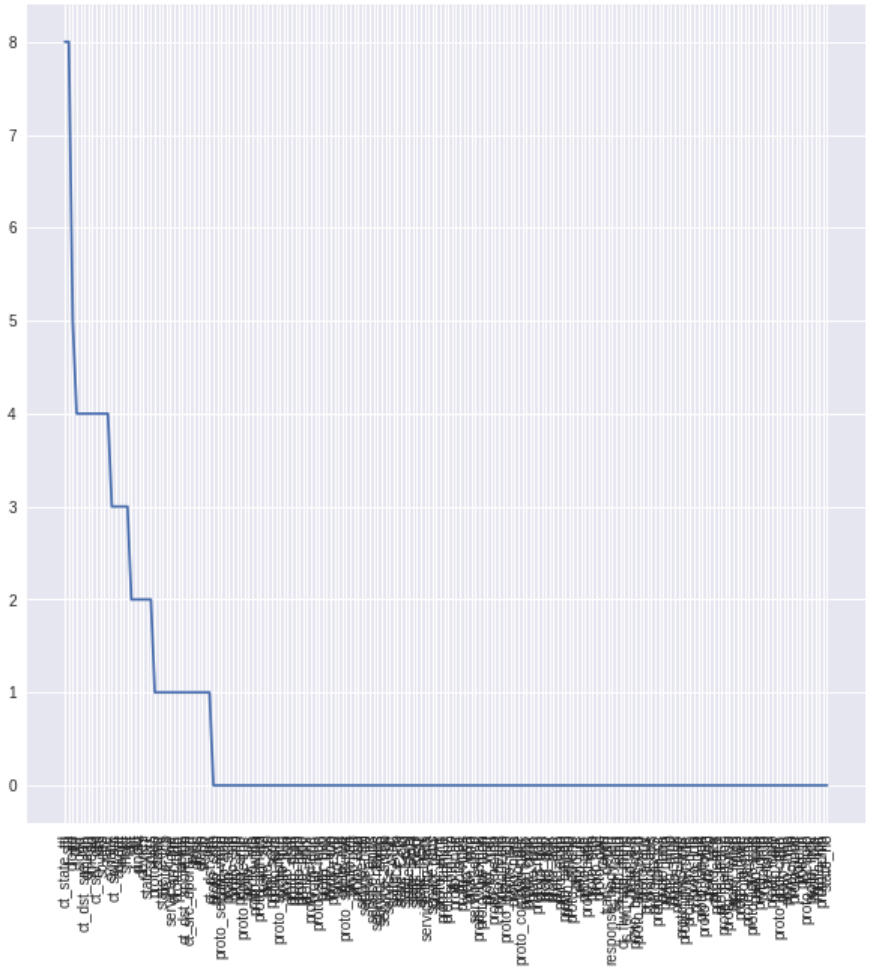
```
from sklearn.svm import LinearSVC
start_time = time.time()

pred_now, acc_svc, acc_cv_svc, svc= fit_algo(LinearSVC()
                                        ,X,Y,10)

svc_time = (time.time() - start_time)

print("Accuracy: %s" % acc_svc)
print("Accuracy of CV: %s" % acc_cv_svc)
print("Execution time: %s" % svc_time)
```

```
Accuracy: 90.26
Accuracy of CV: 89.21
Execution time: 368.23353910446167
```

In our comparative analysis of model performance, we achieved an impressive accuracy of 97.58% with the CNN-LSTM model, demonstrating its effectiveness in capturing complex patterns within the data. The Support Vector Machine (SVM) followed with an accuracy of 89.11%, while the Logistic Regression model closely trailed at 89.05%. In contrast, the Gaussian Naive Bayes model exhibited a significantly lower accuracy of 50.46%. These results highlight the superior performance of the CNN-LSTM architecture in this particular task, underscoring its potential as a robust solution for complex classification problems compared to traditional machine learning methods.



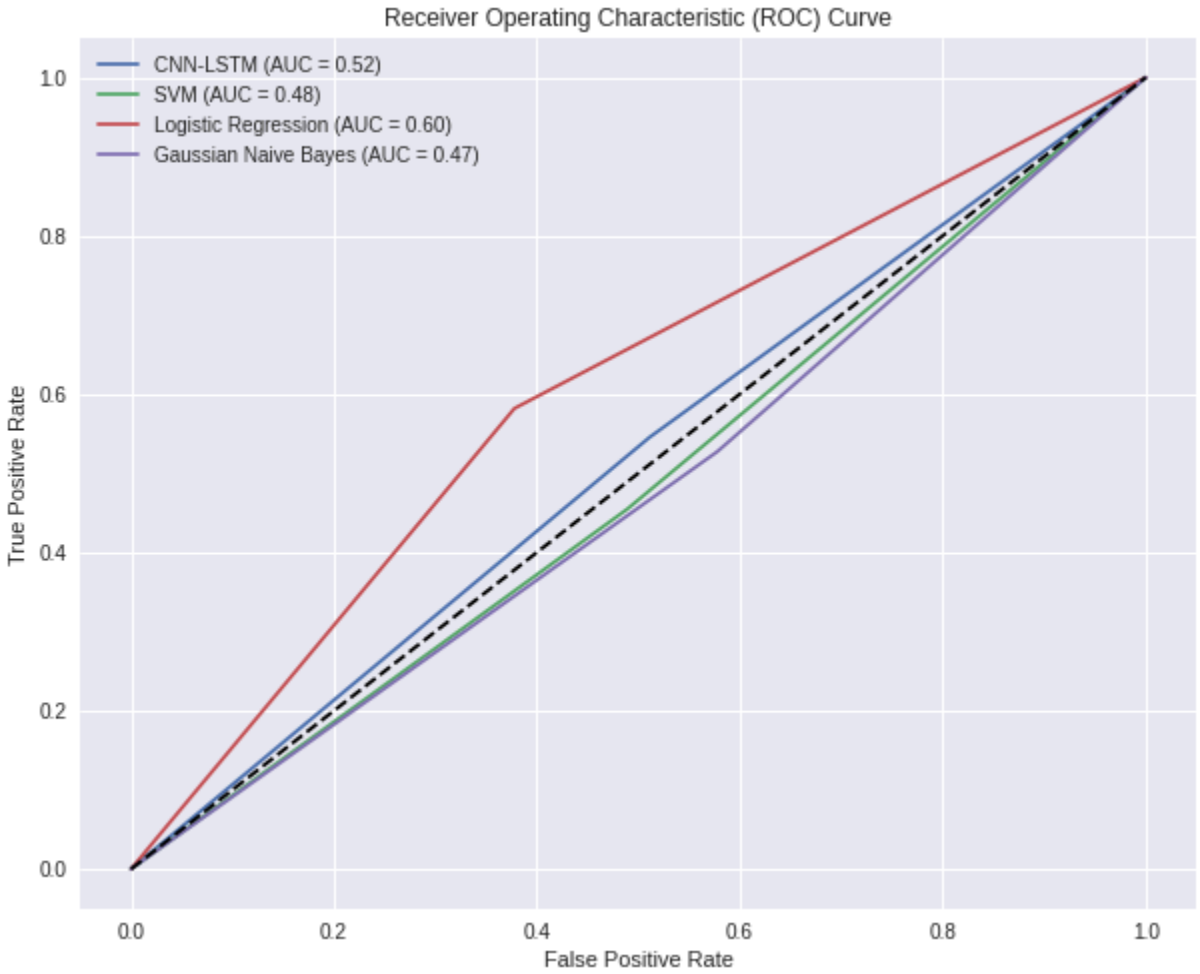
FEATURE IMPORTANCE:



COMPARISON OF MODELS:

S.NO	MODEL	ACCURACY	EXECUTION TIME
1.	CNN-LSTM	97.58%	2540s
2.	SVM	89.11%	349s
3.	LOGISTIC REGRESSION	89.05%	61s
4.	GAUSSIAN NAIVE BAYES	50.46%	10s

AUC - ROC Curve:



Links

- **GitHub Repository:** https://github.com/Leelankitha/ML_End_Sem_Project

Learning Outcomes

1. **Skills Developed:**
    a. **Machine Learning and Deep Learning Skill:** Worked on various machine learning models like SVM, Gaussian Naive Bayes as well deep learning models such as CNN-LSTM.
    b. **Feature Selection and Data Preprocessing:** Various techniques of handling missing values, normalizing dataset, scaling features to train the model.
    c. **Model Evaluation and Comparison:** Learned how to evaluate models using metrics like accuracy, precision, recall, F1-score & the ROC-AUC (Area Under The ROC curve), Confusion Matrix.
    d. **Python Programming:** Advanced Python programming including in the TensorFlow, Keras, Scikit-learn, Pandas, NumPy and Matplotlib libraries to build machine learning models for training and visualization.
2. **Tools and Frameworks Used:**
    a. **Programming Language:** Python was used as the primary programming language.
    b. **Libraries and Frameworks:** Employed TensorFlow and Keras for deep learning models (CNN-LSTM), Scikit-learn for implementing traditional machine learning models (SVM, Gaussian Naive Bayes), Pandas and NumPy for data manipulation, and Matplotlib/Seaborn for data visualization.
    c. **Jupyter Notebook/Google Colab:** Used these platforms for code development, experimentation, and documentation of the machine learning workflow.
    d. **Version Control:** Managed code and project changes using Git/GitHub for collaborative development and version tracking.
3. **Dataset Used:**
    a. **UNSW-NB15 Dataset:** Utilized the UNSW-NB15 network intrusion detection dataset, which includes various network traffic features to classify network behaviors as normal or anomalous.
4. **Topics and Concepts Learned:**
    a. **Supervised Learning Algorithms:** Explored classification techniques using SVM and Gaussian Naive Bayes to build traditional machine learning models for network intrusion detection.
    b. **Deep Learning Architecture (CNN-LSTM):** Learned about hybrid deep learning models, specifically combining Convolutional Neural Networks (CNN) for feature extraction and Long Short-Term Memory (LSTM) networks for sequential data processing.
    c. **Data Science Workflow:** Followed the complete workflow from data preprocessing, model building, training, and evaluation to result analysis and model comparison.
    d. **Performance Metrics:** Understood the significance of different evaluation metrics and their implications for the models' ability to detect intrusions accurately.
    e. **Model Optimization Techniques:** Learned to implement early stopping, model checkpoints, and hyperparameter tuning to enhance the training process and avoid overfitting.