%% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:22:11.868655Z","iopub.execute\_input":"2024-12-13T14:22:11.869793Z","iopub.status.idle":"2024-12-13T14:22:24.580436Z","shell.execute\_reply.started":"2024-12-13T14:22:11.869734Z","shell.execute\_reply":"2024-12-13T14:22:24.579032Z"},"jupyter":{"outputs\_hidden":false}}

pip install dask

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:22:24.583118Z","iopub.execute\_input":"2024-12-13T14:22:24.583535Z","iopub.status.idle":"2024-12-13T14:22:30.556491Z","shell.execute\_reply.started":"2024-12-13T14:22:24.583496Z","shell.execute\_reply":"2024-12-13T14:22:30.555358Z"},"jupyter":{"outputs\_hidden":false}}

import dask.dataframe as dd

# Specify the data type for the problematic column

dtype\_dict = {'SimillarHTTP': 'object'}

# Load the dataset with specified dtype

file\_path = "/kaggle/input/cic-ddos2019-30gb-full-dataset-csv-files/01-12/DrDoS\_DNS.csv"

df = dd.read\_csv(file\_path, dtype=dtype\_dict)

# Display the first few rows

print(df.head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:22:30.558248Z","iopub.execute\_input":"2024-12-13T14:22:30.558740Z","iopub.status.idle":"2024-12-13T14:23:18.839470Z","shell.execute\_reply.started":"2024-12-13T14:22:30.558689Z","shell.execute\_reply":"2024-12-13T14:23:18.838113Z"},"jupyter":{"outputs\_hidden":false}}

missing\_values = df.isnull().sum().compute()

print(missing\_values[missing\_values > 0])

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:23:18.841977Z","iopub.execute\_input":"2024-12-13T14:23:18.842365Z","iopub.status.idle":"2024-12-13T14:24:10.726056Z","shell.execute\_reply.started":"2024-12-13T14:23:18.842330Z","shell.execute\_reply":"2024-12-13T14:24:10.724737Z"},"jupyter":{"outputs\_hidden":false}}

summary = df.describe().compute()

print(summary)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:24:10.727517Z","iopub.execute\_input":"2024-12-13T14:24:10.727874Z","iopub.status.idle":"2024-12-13T14:24:10.734259Z","shell.execute\_reply.started":"2024-12-13T14:24:10.727841Z","shell.execute\_reply":"2024-12-13T14:24:10.732986Z"},"jupyter":{"outputs\_hidden":false}}

# List all columns

print(df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:24:10.735657Z","iopub.execute\_input":"2024-12-13T14:24:10.736048Z","iopub.status.idle":"2024-12-13T14:24:55.757421Z","shell.execute\_reply.started":"2024-12-13T14:24:10.736014Z","shell.execute\_reply":"2024-12-13T14:24:55.756142Z"},"jupyter":{"outputs\_hidden":false}}

# Strip leading and trailing whitespace from column names

df.columns = df.columns.str.strip()

# Retry accessing the 'Label' column

label\_distribution = df['Label'].value\_counts().compute()

print(label\_distribution)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:24:55.759091Z","iopub.execute\_input":"2024-12-13T14:24:55.759504Z","iopub.status.idle":"2024-12-13T14:25:39.575496Z","shell.execute\_reply.started":"2024-12-13T14:24:55.759464Z","shell.execute\_reply":"2024-12-13T14:25:39.574308Z"},"jupyter":{"outputs\_hidden":false}}

label\_distribution = df['Label'].value\_counts().compute()

print(label\_distribution)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:25:39.577295Z","iopub.execute\_input":"2024-12-13T14:25:39.577641Z","iopub.status.idle":"2024-12-13T14:26:23.674939Z","shell.execute\_reply.started":"2024-12-13T14:25:39.577608Z","shell.execute\_reply":"2024-12-13T14:26:23.673622Z"},"jupyter":{"outputs\_hidden":false}}

df['Timestamp'] = dd.to\_datetime(df['Timestamp'], errors='coerce')

# Extract hour and analyze attack frequency

df['Hour'] = df['Timestamp'].dt.hour

hourly\_distribution = df.groupby('Hour').size().compute()

print(hourly\_distribution)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:26:23.676304Z","iopub.execute\_input":"2024-12-13T14:26:23.676656Z","iopub.status.idle":"2024-12-13T14:32:55.825415Z","shell.execute\_reply.started":"2024-12-13T14:26:23.676622Z","shell.execute\_reply":"2024-12-13T14:32:55.823990Z"},"jupyter":{"outputs\_hidden":false}}

labels = df['Label'].unique().compute()

for label in labels:

filtered\_data = df[df['Label'] == label]

output\_path = f"{label}\_chunk.csv"

filtered\_data.compute().to\_csv(output\_path, index=False)

print(f"Saved {output\_path}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:32:55.829072Z","iopub.execute\_input":"2024-12-13T14:32:55.829521Z","iopub.status.idle":"2024-12-13T14:32:56.107493Z","shell.execute\_reply.started":"2024-12-13T14:32:55.829483Z","shell.execute\_reply":"2024-12-13T14:32:56.106347Z"},"jupyter":{"outputs\_hidden":false}}

import matplotlib.pyplot as plt

# Plot hourly distribution

hourly\_distribution.plot(kind='bar')

plt.xlabel('Hour of Day')

plt.ylabel('Frequency')

plt.title('Attack Frequency by Hour')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:32:56.108726Z","iopub.execute\_input":"2024-12-13T14:32:56.109080Z","iopub.status.idle":"2024-12-13T14:33:37.409005Z","shell.execute\_reply.started":"2024-12-13T14:32:56.109047Z","shell.execute\_reply":"2024-12-13T14:33:37.407772Z"},"jupyter":{"outputs\_hidden":false}}

missing\_values = df.isnull().sum().compute()

print("Missing Values:\n", missing\_values[missing\_values > 0])

# %% [code] {"execution":{"iopub.status.busy":"2024-12-13T14:33:37.410701Z","iopub.execute\_input":"2024-12-13T14:33:37.411194Z"},"jupyter":{"outputs\_hidden":false}}

summary = df.describe().compute()

print(summary)

# %% [code] {"jupyter":{"outputs\_hidden":false}}

print(df.dtypes)

# %% [code] {"jupyter":{"outputs\_hidden":false}}

df['Hour'] = dd.to\_datetime(df['Timestamp'], errors='coerce').dt.hour

# %% [code] {"jupyter":{"outputs\_hidden":false}}

import matplotlib.pyplot as plt

# Group by hour and count the number of requests

hourly\_requests = df.groupby('Hour').size().compute()

# Plot the hourly distribution

plt.figure(figsize=(10, 6))

hourly\_requests.plot(kind='bar', color='skyblue')

plt.xlabel('Hour of Day')

plt.ylabel('Number of Requests')

plt.title('Count of Requests Per Hour')

plt.grid(axis='y')

plt.xticks(rotation=0)

plt.tight\_layout()

plt.show()

# %% [code] {"jupyter":{"outputs\_hidden":false}}

import seaborn as sns

import pandas as pd

# Group data by Hour and Label

hourly\_label\_distribution = df.groupby(['Hour', 'Label']).size().compute().reset\_index(name='Count')

# Pivot for heatmap

heatmap\_data = hourly\_label\_distribution.pivot(index='Label', columns='Hour', values='Count').fillna(0)

# Plot the heatmap

plt.figure(figsize=(12, 8))

sns.heatmap(heatmap\_data, cmap='Blues', annot=True, fmt='.0f', linewidths=.5)

plt.xlabel('Hour of Day')

plt.ylabel('Attack Type (Label)')

plt.title('Hourly Activity by Attack Type')

plt.tight\_layout()

plt.show()

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report, accuracy\_score

from sklearn.preprocessing import LabelEncoder

from sklearn.impute import SimpleImputer

import pandas as pd

import numpy as np

# Convert Dask DataFrame to Pandas for modeling

df\_pd = df.compute()

# Define features and target

X = df\_pd.drop(columns=["Timestamp", "Label", "Source IP", "Destination IP"]) # Drop irrelevant columns

y = df\_pd["Label"]

# Encode categorical variables in X

categorical\_columns = X.select\_dtypes(include=["object"]).columns

for col in categorical\_columns:

X[col] = LabelEncoder().fit\_transform(X[col])

# Encode the target variable

y = LabelEncoder().fit\_transform(y)

# Replace inf and -inf with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Impute missing values

num\_imputer = SimpleImputer(strategy="mean") # Impute missing values with mean

X = pd.DataFrame(num\_imputer.fit\_transform(X), columns=X.columns)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train Logistic Regression

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Predictions

y\_pred = log\_reg.predict(X\_test)

# Evaluate the model

print("Logistic Regression - Accuracy:", accuracy\_score(y\_test, y\_pred))

print("Logistic Regression - Classification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=1))

from xgboost import XGBClassifier

# Train XGBoost Classifier

xgb\_clf = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', random\_state=42)

xgb\_clf.fit(X\_train, y\_train)

# Predictions

y\_pred\_xgb = xgb\_clf.predict(X\_test)

# Evaluate

print("XGBoost - Accuracy:", accuracy\_score(y\_test, y\_pred\_xgb))

print("XGBoost - Classification Report:\n", classification\_report(y\_test, y\_pred\_xgb, zero\_division=1))

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, LSTM, Dropout

from sklearn.preprocessing import StandardScaler

# Standardize the data

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Reshape input data to 3D for LSTM (samples, timesteps, features)

X\_reshaped = X\_scaled.reshape((X\_scaled.shape[0], 1, X\_scaled.shape[1]))

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_reshaped, y, test\_size=0.2, random\_state=42)

# Build LSTM model

model = Sequential([

LSTM(64, input\_shape=(X\_train.shape[1], X\_train.shape[2]), return\_sequences=True),

Dropout(0.2),

LSTM(32, return\_sequences=False),

Dropout(0.2),

Dense(1, activation='sigmoid') # Binary classification

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=64, validation\_split=0.2, verbose=1)

# Evaluate the model

loss, accuracy = model.evaluate(X\_test, y\_test, verbose=0)

print(f"LSTM - Test Accuracy: {accuracy}")

# Predict and evaluate

y\_pred = (model.predict(X\_test) > 0.5).astype("int32")

print("LSTM - Classification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=1))

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

import seaborn as sns

import matplotlib.pyplot as plt

# Predictions for Logistic Regression

y\_pred\_lr = logistic\_regression.predict(X\_test)

# Confusion Matrix

conf\_matrix\_lr = confusion\_matrix(y\_test, y\_pred\_lr)

print("Logistic Regression - Confusion Matrix:\n", conf\_matrix\_lr)

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix\_lr, annot=True, fmt='d', cmap='Blues', xticklabels=logistic\_regression.classes\_, yticklabels=logistic\_regression.classes\_)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix - Logistic Regression')

plt.tight\_layout()

plt.show()

# Predictions for XGBoost

y\_pred\_xgb = xgb\_classifier.predict(X\_test)

# Confusion Matrix

conf\_matrix\_xgb = confusion\_matrix(y\_test, y\_pred\_xgb)

print("XGBoost - Confusion Matrix:\n", conf\_matrix\_xgb)

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix\_xgb, annot=True, fmt='d', cmap='Greens', xticklabels=xgb\_classifier.classes\_, yticklabels=xgb\_classifier.classes\_)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix - XGBoost')

plt.tight\_layout()

plt.show()

# Predictions for LSTM

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report, confusion\_matrix

import numpy as np

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_reshaped, y, test\_size=0.2, random\_state=42)

# Build the LSTM model

model = Sequential([

LSTM(64, input\_shape=(X\_train.shape[1], X\_train.shape[2]), return\_sequences=True),

Dropout(0.2),

LSTM(32, return\_sequences=False),

Dropout(0.2),

Dense(1, activation='sigmoid') # Binary classification

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=64, validation\_split=0.2, verbose=1)

# Evaluate the model

loss, accuracy = model.evaluate(X\_test, y\_test, verbose=0)

print(f"LSTM - Test Loss: {loss}, Test Accuracy: {accuracy}")

# Predict on the test set

y\_pred = (model.predict(X\_test) > 0.5).astype("int32")

# Classification Report

print("LSTM - Classification Report:\n", classification\_report(y\_test, y\_pred))

# Confusion Matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:\n", conf\_matrix)

# Visualize Confusion Matrix

import seaborn as sns

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Benign', 'Attack'], yticklabels=['Benign', 'Attack'])

plt.xlabel('Predicted Labels', fontsize=12)

plt.ylabel('Actual Labels', fontsize=12)

plt.title('Confusion Matrix for LSTM', fontsize=14)

plt.show()

**Python codes LDAP**

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:08:26.143490Z","iopub.execute\_input":"2024-12-17T12:08:26.144668Z","iopub.status.idle":"2024-12-17T12:08:38.336878Z","shell.execute\_reply.started":"2024-12-17T12:08:26.144624Z","shell.execute\_reply":"2024-12-17T12:08:38.335618Z"}}

pip install dask

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:08:48.444524Z","iopub.execute\_input":"2024-12-17T12:08:48.445268Z","iopub.status.idle":"2024-12-17T12:08:54.002637Z","shell.execute\_reply.started":"2024-12-17T12:08:48.445224Z","shell.execute\_reply":"2024-12-17T12:08:54.001426Z"}}

import dask.dataframe as dd

# Specify the data type for the problematic column

dtype\_dict = {'SimillarHTTP': 'object'}

# Load the dataset with specified dtype

file\_path = "/kaggle/input/drdos-ldap/DrDoS\_LDAP.csv"

df = dd.read\_csv(file\_path, dtype=dtype\_dict)

# Display the first few rows

print(df.head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:08:59.568843Z","iopub.execute\_input":"2024-12-17T12:08:59.569635Z","iopub.status.idle":"2024-12-17T12:08:59.574926Z","shell.execute\_reply.started":"2024-12-17T12:08:59.569595Z","shell.execute\_reply":"2024-12-17T12:08:59.573884Z"}}

# List all columns

print(df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:09:04.930400Z","iopub.execute\_input":"2024-12-17T12:09:04.930814Z","iopub.status.idle":"2024-12-17T12:09:33.705625Z","shell.execute\_reply.started":"2024-12-17T12:09:04.930783Z","shell.execute\_reply":"2024-12-17T12:09:33.704523Z"}}

import pandas as pd

import numpy as np

# Load the dataset

file\_path = "/kaggle/input/drdos-ldap/DrDoS\_LDAP.csv"

df = pd.read\_csv(file\_path)

# View the first rows and general info

print("Dataset Shape:", df.shape)

print("Dataset Columns:", df.columns)

print(df.head())

# Check for missing values

print("Missing Values:\n", df.isnull().sum())

# Check for unique labels

print("Unique Labels:", df[' Label'].unique())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:09:40.333552Z","iopub.execute\_input":"2024-12-17T12:09:40.334059Z","iopub.status.idle":"2024-12-17T12:09:42.504142Z","shell.execute\_reply.started":"2024-12-17T12:09:40.334026Z","shell.execute\_reply":"2024-12-17T12:09:42.502996Z"}}

# Convert 'SimillarHTTP' to numeric, coercing errors to NaN

df['SimillarHTTP'] = pd.to\_numeric(df['SimillarHTTP'], errors='coerce')

# Fill NaN values safely without inplace=True to avoid chained assignment

df = df.assign(SimillarHTTP=df['SimillarHTTP'].fillna(0))

# Confirm the column is now numeric and clean

print("Updated dtype of SimillarHTTP:", df['SimillarHTTP'].dtype)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:09:47.748186Z","iopub.execute\_input":"2024-12-17T12:09:47.748585Z","iopub.status.idle":"2024-12-17T12:09:47.834039Z","shell.execute\_reply.started":"2024-12-17T12:09:47.748548Z","shell.execute\_reply":"2024-12-17T12:09:47.832930Z"}}

print(df['SimillarHTTP'].describe())

print("Unique values in SimillarHTTP after processing:", df['SimillarHTTP'].unique())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:09:51.430580Z","iopub.execute\_input":"2024-12-17T12:09:51.430977Z","iopub.status.idle":"2024-12-17T12:09:52.802972Z","shell.execute\_reply.started":"2024-12-17T12:09:51.430946Z","shell.execute\_reply":"2024-12-17T12:09:52.802141Z"}}

# Drop unnecessary columns

drop\_columns = ['Unnamed: 0', 'Flow ID', ' Source IP', ' Destination IP', ' Timestamp']

df = df.drop(columns=drop\_columns, axis=1)

# Features and Target

X = df.drop(columns=[' Label']) # Features

y = (df[' Label'] == 'DrDoS\_LDAP').astype(int) # Binary target: 1 for attack, 0 for benign

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T12:09:57.760827Z","iopub.execute\_input":"2024-12-17T12:09:57.761225Z","iopub.status.idle":"2024-12-17T12:09:59.435687Z","shell.execute\_reply.started":"2024-12-17T12:09:57.761192Z","shell.execute\_reply":"2024-12-17T12:09:59.434551Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

# Count of Labels

label\_counts = df[' Label'].value\_counts()

# Plot

plt.figure(figsize=(8, 6))

sns.barplot(x=label\_counts.index, y=label\_counts.values, palette='viridis')

plt.title("Distribution of Attack vs BENIGN")

plt.xlabel("Label")

plt.ylabel("Count")

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T11:15:13.320654Z","iopub.execute\_input":"2024-12-17T11:15:13.321329Z","iopub.status.idle":"2024-12-17T11:20:55.914123Z","shell.execute\_reply.started":"2024-12-17T11:15:13.321289Z","shell.execute\_reply":"2024-12-17T11:20:55.913140Z"}}

import numpy as np

import matplotlib.pyplot as plt

from sklearn.impute import SimpleImputer

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, confusion\_matrix, ConfusionMatrixDisplay

# Step 1: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 2: Impute missing values using the mean

imputer = SimpleImputer(strategy='mean')

X\_imputed = imputer.fit\_transform(X)

# Step 3: Verify no remaining issues

print("Any NaN values left in X?", np.isnan(X\_imputed).any())

print("Any infinite values left in X?", np.isinf(X\_imputed).any())

# Step 4: Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_imputed, y, test\_size=0.3, random\_state=42, stratify=y)

# Step 5: Train Logistic Regression

model = LogisticRegression(max\_iter=1000)

model.fit(X\_train, y\_train)

# Step 6: Predict

y\_pred = model.predict(X\_test)

# Step 7: Evaluate the model

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

# Step 8: Plot the confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=model.classes\_)

# Display the plot

plt.figure(figsize=(8, 6))

disp.plot(cmap=plt.cm.Blues, values\_format='d')

plt.title("Confusion Matrix")

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T11:28:54.411946Z","iopub.execute\_input":"2024-12-17T11:28:54.413446Z","iopub.status.idle":"2024-12-17T11:31:15.928603Z","shell.execute\_reply.started":"2024-12-17T11:28:54.413357Z","shell.execute\_reply":"2024-12-17T11:31:15.927422Z"}}

import pandas as pd

import numpy as np

from sklearn.impute import SimpleImputer

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report, confusion\_matrix, ConfusionMatrixDisplay

import matplotlib.pyplot as plt

# Step 1: Load and clean the dataset

file\_path = "/kaggle/input/drdos-ldap/DrDoS\_LDAP.csv"

# Load dataset

df = pd.read\_csv(file\_path, low\_memory=False)

# Clean column names

df.columns = df.columns.str.strip()

# Drop unnecessary columns

drop\_columns = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

columns\_to\_drop = [col for col in drop\_columns if col in df.columns]

df\_cleaned = df.drop(columns=columns\_to\_drop)

# Encode the Label column: 1 for Attack, 0 for BENIGN

df\_cleaned['Label'] = df\_cleaned['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 2: Select numeric columns only

numeric\_columns = df\_cleaned.select\_dtypes(include=[np.number]).columns

df\_numeric = df\_cleaned[numeric\_columns]

# Separate features and target

X = df\_numeric.drop(columns=['Label']) # Features

y = df\_cleaned['Label'] # Target

# Step 3: Handle missing values

X.replace([np.inf, -np.inf], np.nan, inplace=True)

imputer = SimpleImputer(strategy='mean')

X\_imputed = imputer.fit\_transform(X)

# Step 4: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_imputed, y, test\_size=0.3, random\_state=42, stratify=y)

# Step 5: Train Random Forest Classifier

print("Training Random Forest...")

rf\_model = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_model.fit(X\_train, y\_train)

# Step 6: Predictions

y\_pred = rf\_model.predict(X\_test)

# Step 7: Evaluate the Model

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

# Confusion Matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:\n", conf\_matrix)

# Step 8: Plot Confusion Matrix

plt.figure(figsize=(8, 6))

disp = ConfusionMatrixDisplay(confusion\_matrix=conf\_matrix, display\_labels=['BENIGN (0)', 'Attack (1)'])

disp.plot(cmap='Blues', values\_format='d')

plt.title("Confusion Matrix - Random Forest")

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T11:52:30.566460Z","iopub.execute\_input":"2024-12-17T11:52:30.567071Z","iopub.status.idle":"2024-12-17T11:53:47.919981Z","shell.execute\_reply.started":"2024-12-17T11:52:30.567033Z","shell.execute\_reply":"2024-12-17T11:53:47.918960Z"}}

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from xgboost import XGBClassifier

from sklearn.metrics import classification\_report, confusion\_matrix, ConfusionMatrixDisplay

from sklearn.impute import SimpleImputer

import matplotlib.pyplot as plt

# Load the dataset

file\_path = '/kaggle/input/drdos-ldap/DrDoS\_LDAP.csv'

df = pd.read\_csv(file\_path, low\_memory=False)

# Clean column names by removing spaces

df.columns = df.columns.str.strip()

# Drop unnecessary columns

drop\_columns = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df\_cleaned = df.drop(columns=drop\_columns, errors='ignore')

# Encode the 'Label' column: 1 for Attack, 0 for BENIGN

df\_cleaned['Label'] = df\_cleaned['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Select only numeric columns

numeric\_cols = df\_cleaned.select\_dtypes(include=[np.number]).columns

df\_cleaned = df\_cleaned[numeric\_cols]

# Separate features and target

X = df\_cleaned.drop(columns=['Label'], errors='ignore')

y = df\_cleaned['Label']

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Handle missing values using SimpleImputer

imputer = SimpleImputer(strategy='mean')

X\_imputed = imputer.fit\_transform(X)

# Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_imputed, y, test\_size=0.3, random\_state=42, stratify=y)

# Train XGBoost Classifier

print("Training XGBoost Classifier...")

xgb\_model = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', random\_state=42)

xgb\_model.fit(X\_train, y\_train)

# Predictions

y\_pred = xgb\_model.predict(X\_test)

# Classification Report

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

# Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=['BENIGN', 'ATTACK'])

disp.plot()

plt.title("Confusion Matrix for XGBoost")

plt.show()

# %% [code] {"jupyter":{"outputs\_hidden":false},"execution":{"iopub.status.busy":"2024-12-17T19:12:52.766230Z","iopub.execute\_input":"2024-12-17T19:12:52.766623Z","iopub.status.idle":"2024-12-17T19:41:56.363149Z","shell.execute\_reply.started":"2024-12-17T19:12:52.766584Z","shell.execute\_reply":"2024-12-17T19:41:56.361411Z"}}

# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.callbacks import EarlyStopping

import keras\_tuner as kt

# Step 1: Load and Sample the Dataset

file\_path = '/kaggle/input/drdos-ldap/DrDoS\_LDAP.csv' # Update the path

df = pd.read\_csv(file\_path)

print("Data Loaded Successfully!")

print("Initial shape of data:", df.shape)

# Take a random sample

df\_sample = df.sample(n=452450, random\_state=42)

print("Shape of sampled data:", df\_sample.shape)

# Step 2: Data Preprocessing

# Replace infinite and NaN values

df\_sample.replace([np.inf, -np.inf], np.nan, inplace=True)

df\_sample.dropna(inplace=True)

# Select only numerical columns

X\_sample = df\_sample.select\_dtypes(include=[np.number]).drop(columns=['Unnamed: 0'])

print("Shape after preprocessing:", X\_sample.shape)

# Normalize the data

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X\_sample)

# Step 3: Create Sequences for LSTM

def create\_sequences(data, time\_steps=5):

sequences = []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

return np.array(sequences)

time\_steps = 5

sequences = create\_sequences(X\_scaled, time\_steps)

print("Shape of sequences:", sequences.shape)

# Step 4: Train-Test Split

train\_size = int(0.8 \* len(sequences))

train\_data = sequences[:train\_size]

test\_data = sequences[train\_size:]

print("Training data shape:", train\_data.shape)

print("Testing data shape:", test\_data.shape)

# Step 5: Define the LSTM Model using Keras Tuner

def model\_builder(hp):

model = Sequential()

hp\_units = hp.Int('units', min\_value=16, max\_value=128, step=16)

model.add(LSTM(hp\_units, input\_shape=(time\_steps, X\_scaled.shape[1]), return\_sequences=False))

model.add(Dropout(hp.Float('dropout', min\_value=0.1, max\_value=0.5, step=0.1)))

model.add(Dense(X\_scaled.shape[1]))

model.compile(optimizer='adam', loss='mse')

return model

# Step 6: Hyperparameter Tuning

tuner = kt.RandomSearch(

model\_builder,

objective='val\_loss',

max\_trials=5, # Number of trials for hyperparameter tuning

executions\_per\_trial=1,

directory='tuning\_dir',

project\_name='lstm\_tuning'

)

# Step 7: Run Hyperparameter Tuning

batch\_size = 128

tuner.search(train\_data, train\_data[:, -1, :],

validation\_data=(test\_data, test\_data[:, -1, :]),

epochs=5, batch\_size=batch\_size)

# Get the best hyperparameters

best\_hps = tuner.get\_best\_hyperparameters(num\_trials=1)[0]

print(f"Best Units: {best\_hps.get('units')}, Best Dropout: {best\_hps.get('dropout')}")

# Step 8: Build the Final Model with Best Hyperparameters

final\_model = tuner.hypermodel.build(best\_hps)

early\_stop = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

history = final\_model.fit(

train\_data, train\_data[:, -1, :],

validation\_data=(test\_data, test\_data[:, -1, :]),

batch\_size=batch\_size,

epochs=20,

callbacks=[early\_stop],

verbose=1

)

# Step 9: Evaluate Reconstruction Error

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data)

reconstruction\_error = np.mean(np.square(data[:, -1, :] - predictions), axis=1)

return reconstruction\_error

# Calculate reconstruction errors

train\_errors = calculate\_reconstruction\_error(train\_data, final\_model)

test\_errors = calculate\_reconstruction\_error(test\_data, final\_model)

# Determine the threshold

threshold = np.percentile(train\_errors, 95)

print("Reconstruction error threshold:", threshold)

# Step 10: Plot Reconstruction Errors

plt.figure(figsize=(10, 6))

plt.hist(test\_errors, bins=50, color='blue', alpha=0.7, label="Test Error")

plt.axvline(x=threshold, color='red', linestyle='--', label="Threshold")

plt.title("Reconstruction Error Distribution")

plt.xlabel("Reconstruction Error")

plt.ylabel("Frequency")

plt.legend()

plt.show()

# Step 11: Anomaly Detection

anomalies = test\_errors > threshold

num\_anomalies = np.sum(anomalies)

print("Number of anomalies detected:", num\_anomalies)

**Python codes MSSQL**

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T14:05:44.576987Z","iopub.execute\_input":"2024-12-17T14:05:44.577375Z","iopub.status.idle":"2024-12-17T14:06:49.729225Z","shell.execute\_reply.started":"2024-12-17T14:05:44.577328Z","shell.execute\_reply":"2024-12-17T14:06:49.728093Z"}}

import pandas as pd

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-mssql/DrDoS\_MSSQL.csv" # Replace with the correct path

df = pd.read\_csv(file\_path)

# Step 2: Inspect the columns

df.columns = df.columns.str.strip() # Clean column names

print("Columns in the dataset:\n", df.columns)

# Step 3: Preview the data

print("\nFirst 5 rows of the data:")

print(df.head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T14:08:01.995581Z","iopub.execute\_input":"2024-12-17T14:08:01.996129Z","iopub.status.idle":"2024-12-17T14:08:07.864002Z","shell.execute\_reply.started":"2024-12-17T14:08:01.996081Z","shell.execute\_reply":"2024-12-17T14:08:07.862863Z"}}

import pandas as pd

import matplotlib.pyplot as plt

# Step 1: Convert 'Timestamp' to datetime format

df['Timestamp'] = pd.to\_datetime(df['Timestamp'], errors='coerce')

# Step 2: Sort the data by 'Timestamp'

df = df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Step 3: Plot 'Flow Duration' over time

plt.figure(figsize=(12, 6))

plt.plot(df['Timestamp'], df['Flow Duration'], color='blue', linewidth=1)

plt.xlabel("Timestamp")

plt.ylabel("Flow Duration")

plt.title("Flow Duration Over Time")

plt.grid()

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T14:16:28.933165Z","iopub.execute\_input":"2024-12-17T14:16:28.933982Z","iopub.status.idle":"2024-12-17T14:16:50.628974Z","shell.execute\_reply.started":"2024-12-17T14:16:28.933939Z","shell.execute\_reply":"2024-12-17T14:16:50.627780Z"}}

from statsmodels.tsa.seasonal import seasonal\_decompose

from sklearn.preprocessing import MinMaxScaler

# Ensure Timestamp is datetime and set it as the index

df['Timestamp'] = pd.to\_datetime(df['Timestamp'], errors='coerce')

df = df.dropna(subset=['Timestamp']) # Drop rows with invalid timestamps

df = df.sort\_values(by='Timestamp').reset\_index(drop=True)

df.set\_index('Timestamp', inplace=True)

# Focus on Flow Duration

flow\_duration = df['Flow Duration']

# Plot the original Flow Duration time series

plt.figure(figsize=(12, 6))

plt.plot(flow\_duration, color='blue', label='Flow Duration')

plt.title('Original Flow Duration Over Time')

plt.xlabel('Timestamp')

plt.ylabel('Flow Duration')

plt.legend()

plt.show()

# Step 2: Decompose the time series to see trend and seasonality

decomposition = seasonal\_decompose(flow\_duration, model='additive', period=100) # Adjust period as needed

# Plot decomposed components

plt.figure(figsize=(12, 8))

decomposition.plot()

plt.suptitle('Decomposition of Flow Duration Time Series', fontsize=14)

plt.tight\_layout()

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T18:03:30.461002Z","iopub.execute\_input":"2024-12-17T18:03:30.461700Z","iopub.status.idle":"2024-12-17T18:08:50.380942Z","shell.execute\_reply.started":"2024-12-17T18:03:30.461632Z","shell.execute\_reply":"2024-12-17T18:08:50.379572Z"}}

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from sklearn.model\_selection import train\_test\_split

import random

# Step 1: Load and Sample the Data

file\_path = '/kaggle/input/drdos-mssql/DrDoS\_MSSQL.csv' # Replace with your file path

print("Loading data...")

df = pd.read\_csv(file\_path)

# Reduce dataset size by sampling 10% of the rows

df\_sampled = df.sample(frac=0.1, random\_state=42)

print(f"Shape of sampled data: {df\_sampled.shape}")

# Drop unnecessary columns and retain numerical features

X = df\_sampled.select\_dtypes(include=[np.number]).drop(columns=['Unnamed: 0'], errors='ignore')

# Handle missing or infinite values

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X = X.dropna()

print("Any NaN values left?", X.isnull().any().any())

print("Any infinite values left?", np.isinf(X.values).any())

# Normalize the data

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

print("Data scaling completed!")

# Step 2: Create Sequences for LSTM

time\_steps = 5 # Reduce time steps to save memory

batch\_size = 32 # Smaller batch size to reduce memory usage

def create\_sequences(data, time\_steps=5):

sequences = []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

return np.array(sequences)

# Split data into training and testing

X\_train, X\_test = train\_test\_split(X\_scaled, test\_size=0.2, shuffle=False)

# Create sequences for LSTM

X\_train\_seq = create\_sequences(X\_train, time\_steps)

X\_test\_seq = create\_sequences(X\_test, time\_steps)

print(f"Shape of training sequences: {X\_train\_seq.shape}")

print(f"Shape of testing sequences: {X\_test\_seq.shape}")

# Step 3: Build a Simplified LSTM Autoencoder

model = Sequential([

LSTM(32, activation='relu', input\_shape=(time\_steps, X\_scaled.shape[1])),

Dropout(0.2),

Dense(X\_scaled.shape[1])

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model in Smaller Batches

history = model.fit(

X\_train\_seq, X\_train\_seq[:, -1, :], # Use the last step for reconstruction

validation\_data=(X\_test\_seq, X\_test\_seq[:, -1, :]),

epochs=5,

batch\_size=batch\_size,

verbose=1

)

# Step 5: Evaluate Reconstruction Error

def calculate\_reconstruction\_error(data, model):

reconstructed = model.predict(data, verbose=0)

error = np.mean(np.abs(reconstructed - data[:, -1, :]), axis=1)

return error

train\_error = calculate\_reconstruction\_error(X\_train\_seq, model)

test\_error = calculate\_reconstruction\_error(X\_test\_seq, model)

# Set threshold for anomalies (95th percentile)

threshold = np.percentile(train\_error, 95)

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_error > threshold

num\_anomalies = np.sum(test\_anomalies)

print("Number of anomalies detected:", num\_anomalies)

# Step 6: Plot Results

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

plt.hist(test\_error, bins=50, color='blue', alpha=0.7, label='Test Error')

plt.axvline(threshold, color='red', linestyle='dashed', linewidth=2, label='Threshold')

plt.title('Reconstruction Error Distribution')

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.legend()

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-17T18:20:42.555956Z","iopub.execute\_input":"2024-12-17T18:20:42.558237Z","iopub.status.idle":"2024-12-17T18:44:14.015620Z","shell.execute\_reply.started":"2024-12-17T18:20:42.558184Z","shell.execute\_reply":"2024-12-17T18:44:14.014204Z"}}

# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.callbacks import EarlyStopping

import keras\_tuner as kt

# Step 1: Load and Sample the Dataset

file\_path = '/kaggle/input/drdos-mssql/DrDoS\_MSSQL.csv' # Update the path

df = pd.read\_csv(file\_path)

print("Data Loaded Successfully!")

print("Initial shape of data:", df.shape)

# Take a random sample

df\_sample = df.sample(n=452450, random\_state=42)

print("Shape of sampled data:", df\_sample.shape)

# Step 2: Data Preprocessing

# Replace infinite and NaN values

df\_sample.replace([np.inf, -np.inf], np.nan, inplace=True)

df\_sample.dropna(inplace=True)

# Select only numerical columns

X\_sample = df\_sample.select\_dtypes(include=[np.number]).drop(columns=['Unnamed: 0'])

print("Shape after preprocessing:", X\_sample.shape)

# Normalize the data

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X\_sample)

# Step 3: Create Sequences for LSTM

def create\_sequences(data, time\_steps=5):

sequences = []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

return np.array(sequences)

time\_steps = 5

sequences = create\_sequences(X\_scaled, time\_steps)

print("Shape of sequences:", sequences.shape)

# Step 4: Train-Test Split

train\_size = int(0.8 \* len(sequences))

train\_data = sequences[:train\_size]

test\_data = sequences[train\_size:]

print("Training data shape:", train\_data.shape)

print("Testing data shape:", test\_data.shape)

# Step 5: Define the LSTM Model using Keras Tuner

def model\_builder(hp):

model = Sequential()

hp\_units = hp.Int('units', min\_value=16, max\_value=128, step=16)

model.add(LSTM(hp\_units, input\_shape=(time\_steps, X\_scaled.shape[1]), return\_sequences=False))

model.add(Dropout(hp.Float('dropout', min\_value=0.1, max\_value=0.5, step=0.1)))

model.add(Dense(X\_scaled.shape[1]))

model.compile(optimizer='adam', loss='mse')

return model

# Step 6: Hyperparameter Tuning

tuner = kt.RandomSearch(

model\_builder,

objective='val\_loss',

max\_trials=5, # Number of trials for hyperparameter tuning

executions\_per\_trial=1,

directory='tuning\_dir',

project\_name='lstm\_tuning'

)

# Step 7: Run Hyperparameter Tuning

batch\_size = 128

tuner.search(train\_data, train\_data[:, -1, :],

validation\_data=(test\_data, test\_data[:, -1, :]),

epochs=5, batch\_size=batch\_size)

# Get the best hyperparameters

best\_hps = tuner.get\_best\_hyperparameters(num\_trials=1)[0]

print(f"Best Units: {best\_hps.get('units')}, Best Dropout: {best\_hps.get('dropout')}")

# Step 8: Build the Final Model with Best Hyperparameters

final\_model = tuner.hypermodel.build(best\_hps)

early\_stop = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

history = final\_model.fit(

train\_data, train\_data[:, -1, :],

validation\_data=(test\_data, test\_data[:, -1, :]),

batch\_size=batch\_size,

epochs=20,

callbacks=[early\_stop],

verbose=1

)

# Step 9: Evaluate Reconstruction Error

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data)

reconstruction\_error = np.mean(np.square(data[:, -1, :] - predictions), axis=1)

return reconstruction\_error

# Calculate reconstruction errors

train\_errors = calculate\_reconstruction\_error(train\_data, final\_model)

test\_errors = calculate\_reconstruction\_error(test\_data, final\_model)

# Determine the threshold

threshold = np.percentile(train\_errors, 95)

print("Reconstruction error threshold:", threshold)

# Step 10: Plot Reconstruction Errors

plt.figure(figsize=(10, 6))

plt.hist(test\_errors, bins=50, color='blue', alpha=0.7, label="Test Error")

plt.axvline(x=threshold, color='red', linestyle='--', label="Threshold")

plt.title("Reconstruction Error Distribution")

plt.xlabel("Reconstruction Error")

plt.ylabel("Frequency")

plt.legend()

plt.show()

# Step 11: Anomaly Detection

anomalies = test\_errors > threshold

num\_anomalies = np.sum(anomalies)

print("Number of anomalies detected:", num\_anomalies)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:18:25.563465Z","iopub.execute\_input":"2024-12-19T15:18:25.563848Z","iopub.status.idle":"2024-12-19T15:19:46.358015Z","shell.execute\_reply.started":"2024-12-19T15:18:25.563814Z","shell.execute\_reply":"2024-12-19T15:19:46.356752Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-mssql/DrDoS\_MSSQL.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:20:12.039217Z","iopub.execute\_input":"2024-12-19T15:20:12.039642Z","iopub.status.idle":"2024-12-19T15:21:42.628316Z","shell.execute\_reply.started":"2024-12-19T15:20:12.039584Z","shell.execute\_reply":"2024-12-19T15:21:42.627131Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-mssql/DrDoS\_MSSQL.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:21:49.441147Z","iopub.execute\_input":"2024-12-19T15:21:49.441518Z","iopub.status.idle":"2024-12-19T15:23:14.122433Z","shell.execute\_reply.started":"2024-12-19T15:21:49.441488Z","shell.execute\_reply":"2024-12-19T15:23:14.120914Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:23:20.866688Z","iopub.execute\_input":"2024-12-19T15:23:20.867119Z","iopub.status.idle":"2024-12-19T15:23:21.757889Z","shell.execute\_reply.started":"2024-12-19T15:23:20.867088Z","shell.execute\_reply":"2024-12-19T15:23:21.756455Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:30:18.113222Z","iopub.execute\_input":"2024-12-19T15:30:18.113903Z","iopub.status.idle":"2024-12-19T15:30:18.543108Z","shell.execute\_reply.started":"2024-12-19T15:30:18.113859Z","shell.execute\_reply":"2024-12-19T15:30:18.541644Z"}}

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:27:59.275591Z","iopub.execute\_input":"2024-12-19T15:27:59.276241Z","iopub.status.idle":"2024-12-19T15:27:59.757899Z","shell.execute\_reply.started":"2024-12-19T15:27:59.276208Z","shell.execute\_reply":"2024-12-19T15:27:59.756719Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:28:14.081109Z","iopub.execute\_input":"2024-12-19T15:28:14.081463Z","iopub.status.idle":"2024-12-19T15:28:14.088537Z","shell.execute\_reply.started":"2024-12-19T15:28:14.081436Z","shell.execute\_reply":"2024-12-19T15:28:14.086711Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:28:37.494276Z","iopub.execute\_input":"2024-12-19T15:28:37.494640Z","iopub.status.idle":"2024-12-19T15:28:38.033500Z","shell.execute\_reply.started":"2024-12-19T15:28:37.494590Z","shell.execute\_reply":"2024-12-19T15:28:38.032200Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:28:43.486034Z","iopub.execute\_input":"2024-12-19T15:28:43.486409Z","iopub.status.idle":"2024-12-19T15:28:44.201829Z","shell.execute\_reply.started":"2024-12-19T15:28:43.486378Z","shell.execute\_reply":"2024-12-19T15:28:44.200219Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:28:57.000453Z","iopub.execute\_input":"2024-12-19T15:28:57.000927Z","iopub.status.idle":"2024-12-19T15:28:57.595516Z","shell.execute\_reply.started":"2024-12-19T15:28:57.000893Z","shell.execute\_reply":"2024-12-19T15:28:57.594274Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:29:10.801899Z","iopub.execute\_input":"2024-12-19T15:29:10.802254Z","iopub.status.idle":"2024-12-19T15:29:14.040950Z","shell.execute\_reply.started":"2024-12-19T15:29:10.802227Z","shell.execute\_reply":"2024-12-19T15:29:14.038758Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:29:28.917432Z","iopub.execute\_input":"2024-12-19T15:29:28.917849Z","iopub.status.idle":"2024-12-19T15:29:29.125413Z","shell.execute\_reply.started":"2024-12-19T15:29:28.917817Z","shell.execute\_reply":"2024-12-19T15:29:29.124209Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-19T15:29:44.100386Z","iopub.execute\_input":"2024-12-19T15:29:44.100930Z","iopub.status.idle":"2024-12-19T15:29:44.813280Z","shell.execute\_reply.started":"2024-12-19T15:29:44.100881Z","shell.execute\_reply":"2024-12-19T15:29:44.811982Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

**Python codes NetBIOS**

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:22:39.513758Z","iopub.execute\_input":"2024-12-18T10:22:39.514357Z","iopub.status.idle":"2024-12-18T10:23:27.509976Z","shell.execute\_reply.started":"2024-12-18T10:22:39.514290Z","shell.execute\_reply":"2024-12-18T10:23:27.508188Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-netbios/DrDoS\_NetBIOS.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:26:41.172656Z","iopub.execute\_input":"2024-12-18T10:26:41.173108Z","iopub.status.idle":"2024-12-18T10:26:41.179531Z","shell.execute\_reply.started":"2024-12-18T10:26:41.173068Z","shell.execute\_reply":"2024-12-18T10:26:41.178454Z"}}

# Check the column names

print("Columns in the dataset:", sampled\_df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:28:12.615532Z","iopub.execute\_input":"2024-12-18T10:28:12.615997Z","iopub.status.idle":"2024-12-18T10:28:13.642664Z","shell.execute\_reply.started":"2024-12-18T10:28:12.615958Z","shell.execute\_reply":"2024-12-18T10:28:13.641539Z"}}

# Step 3: Clean column names by stripping spaces

sampled\_df.columns = sampled\_df.columns.str.strip()

# Verify cleaned column names

print("Cleaned Columns:", sampled\_df.columns)

# Step 4: Convert 'Timestamp' column to datetime and sort the data

sampled\_df['Timestamp'] = pd.to\_datetime(sampled\_df['Timestamp'], errors='coerce')

sampled\_df = sampled\_df.dropna(subset=['Timestamp']) # Drop invalid timestamps

sampled\_df = sampled\_df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Verify the cleaned and sorted data

print("Data sorted by Timestamp:")

print(sampled\_df[['Timestamp']].head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:30:33.460654Z","iopub.execute\_input":"2024-12-18T10:30:33.461089Z","iopub.status.idle":"2024-12-18T10:30:33.482633Z","shell.execute\_reply.started":"2024-12-18T10:30:33.461050Z","shell.execute\_reply":"2024-12-18T10:30:33.481198Z"}}

# Check for infinite values

print("Any infinite values in X:", np.isinf(X).any().any())

# Check for very large values

print("Max value in X:", np.max(X.values))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:30:48.121310Z","iopub.execute\_input":"2024-12-18T10:30:48.121739Z","iopub.status.idle":"2024-12-18T10:30:48.184276Z","shell.execute\_reply.started":"2024-12-18T10:30:48.121697Z","shell.execute\_reply":"2024-12-18T10:30:48.183084Z"}}

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Optionally, cap large values (if necessary)

X = np.clip(X, -1e6, 1e6) # Adjust the range as per your dataset

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:31:11.253177Z","iopub.execute\_input":"2024-12-18T10:31:11.254337Z","iopub.status.idle":"2024-12-18T10:31:11.300625Z","shell.execute\_reply.started":"2024-12-18T10:31:11.254271Z","shell.execute\_reply":"2024-12-18T10:31:11.299461Z"}}

# Fill NaN with column mean (or other strategies like median)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:31:21.658941Z","iopub.execute\_input":"2024-12-18T10:31:21.659342Z","iopub.status.idle":"2024-12-18T10:31:21.707595Z","shell.execute\_reply.started":"2024-12-18T10:31:21.659307Z","shell.execute\_reply":"2024-12-18T10:31:21.706452Z"}}

# Normalize features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:31:33.942518Z","iopub.execute\_input":"2024-12-18T10:31:33.942961Z","iopub.status.idle":"2024-12-18T10:31:34.054763Z","shell.execute\_reply.started":"2024-12-18T10:31:33.942922Z","shell.execute\_reply":"2024-12-18T10:31:34.053470Z"}}

from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Step 1: Select relevant features for modeling

features = [

'Flow Duration', 'Total Fwd Packets', 'Total Backward Packets',

'Total Length of Fwd Packets', 'Total Length of Bwd Packets',

'Flow Bytes/s', 'Flow Packets/s', 'Active Mean', 'Idle Mean'

]

X = sampled\_df[features]

# Step 2: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 3: Fill missing values with column means

X.fillna(X.mean(), inplace=True)

# Step 4: Normalize the features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:34:09.023144Z","iopub.execute\_input":"2024-12-18T10:34:09.023746Z","iopub.status.idle":"2024-12-18T10:50:05.964139Z","shell.execute\_reply.started":"2024-12-18T10:34:09.023699Z","shell.execute\_reply":"2024-12-18T10:50:05.962366Z"}}

import numpy as np

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dropout, Dense

# Step 1: Create Sequences for LSTM

def create\_sequences(data, time\_steps=10):

sequences, targets = [], []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

targets.append(data[i + time\_steps])

return np.array(sequences), np.array(targets)

# Time step for LSTM

time\_steps = 10

# Create sequences from the scaled data

X\_sequences, y\_sequences = create\_sequences(X\_scaled, time\_steps)

# Step 2: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_sequences, y\_sequences, test\_size=0.2, random\_state=42

)

print("Shape of training sequences:", X\_train.shape)

print("Shape of testing sequences:", X\_test.shape)

# Step 3: Define the LSTM Model

model = Sequential([

LSTM(64, activation='tanh', input\_shape=(time\_steps, X\_train.shape[2])),

Dropout(0.2),

Dense(X\_train.shape[2], activation='linear') # Output layer matches feature size

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_test, y\_test),

epochs=20,

batch\_size=64,

verbose=1

)

# Step 5: Evaluate Reconstruction Errors

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data, verbose=0)

errors = np.mean(np.abs(data - predictions), axis=1) # Mean Absolute Error per sequence

return errors

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training data

threshold = np.percentile(train\_errors, 95) # 95th percentile

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_errors > threshold

print("Number of anomalies detected:", np.sum(test\_anomalies))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:54:42.609440Z","iopub.execute\_input":"2024-12-18T10:54:42.609865Z","iopub.status.idle":"2024-12-18T10:54:42.616018Z","shell.execute\_reply.started":"2024-12-18T10:54:42.609830Z","shell.execute\_reply":"2024-12-18T10:54:42.614759Z"}}

def calculate\_reconstruction\_error(data, model):

"""

Calculate reconstruction error for LSTM autoencoder.

:param data: Input data (3D: samples, timesteps, features)

:param model: Trained LSTM autoencoder

:return: Reconstruction errors (1D array)

"""

predictions = model.predict(data, verbose=0) # Shape: (samples, features)

# Use only the last timestep of the input sequences

data\_last\_step = data[:, -1, :] # Shape: (samples, features)

# Calculate Mean Absolute Error (MAE) per sequence

errors = np.mean(np.abs(data\_last\_step - predictions), axis=1)

return errors

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:54:53.491638Z","iopub.execute\_input":"2024-12-18T10:54:53.492083Z","iopub.status.idle":"2024-12-18T10:55:37.439387Z","shell.execute\_reply.started":"2024-12-18T10:54:53.492045Z","shell.execute\_reply":"2024-12-18T10:55:37.438187Z"}}

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training errors

threshold = np.percentile(train\_errors, 95) # e.g., 95th percentile

# Identify anomalies in test set

test\_anomalies = test\_errors > threshold

print(f"Anomaly detection threshold: {threshold}")

print(f"Number of anomalies detected: {np.sum(test\_anomalies)}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T10:56:59.178361Z","iopub.execute\_input":"2024-12-18T10:56:59.178810Z","iopub.status.idle":"2024-12-18T10:56:59.715848Z","shell.execute\_reply.started":"2024-12-18T10:56:59.178769Z","shell.execute\_reply":"2024-12-18T10:56:59.714741Z"}}

import matplotlib.pyplot as plt

plt.hist(train\_errors, bins=50, alpha=0.6, label='Train Errors')

plt.hist(test\_errors, bins=50, alpha=0.6, label='Test Errors')

plt.axvline(x=threshold, color='r', linestyle='--', label='Threshold')

plt.legend()

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.title('Reconstruction Error Distribution')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:14:36.082130Z","iopub.execute\_input":"2024-12-18T11:14:36.082655Z","iopub.status.idle":"2024-12-18T11:15:21.448570Z","shell.execute\_reply.started":"2024-12-18T11:14:36.082612Z","shell.execute\_reply":"2024-12-18T11:15:21.447302Z"}}

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-netbios/DrDoS\_NetBIOS.csv"

df = pd.read\_csv(file\_path)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df = df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column (Assume 'Label' contains attack type)

df['Label'] = df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 5: Separate Features and Target

X = df.drop(columns=['Label'])

y = df['Label']

print("Cleaned dataset shape:", X.shape, y.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:15:38.953377Z","iopub.execute\_input":"2024-12-18T11:15:38.953892Z","iopub.status.idle":"2024-12-18T11:16:55.085941Z","shell.execute\_reply.started":"2024-12-18T11:15:38.953850Z","shell.execute\_reply":"2024-12-18T11:16:55.084577Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-netbios/DrDoS\_NetBIOS.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Random Sampling (e.g., 10% of the data)

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for desired percentage

# Step 4: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 5: Encode Target Column (Assume 'Label' contains attack type)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 6: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 7: Split Data into Training and Test Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

# Final Output Shapes

print("Randomly sampled dataset shape:", sampled\_df.shape)

print("Training set shape:", X\_train.shape, y\_train.shape)

print("Testing set shape:", X\_test.shape, y\_test.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:26:58.368354Z","iopub.execute\_input":"2024-12-18T11:26:58.369561Z","iopub.status.idle":"2024-12-18T11:28:13.192096Z","shell.execute\_reply.started":"2024-12-18T11:26:58.369489Z","shell.execute\_reply":"2024-12-18T11:28:13.190860Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-netbios/DrDoS\_NetBIOS.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:42:56.390820Z","iopub.execute\_input":"2024-12-18T11:42:56.391546Z","iopub.status.idle":"2024-12-18T11:44:12.778062Z","shell.execute\_reply.started":"2024-12-18T11:42:56.391453Z","shell.execute\_reply":"2024-12-18T11:44:12.776653Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:45:47.013701Z","iopub.execute\_input":"2024-12-18T11:45:47.014179Z","iopub.status.idle":"2024-12-18T11:45:47.812552Z","shell.execute\_reply.started":"2024-12-18T11:45:47.014140Z","shell.execute\_reply":"2024-12-18T11:45:47.811372Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T11:59:48.625565Z","iopub.execute\_input":"2024-12-18T11:59:48.626569Z","iopub.status.idle":"2024-12-18T11:59:49.238750Z","shell.execute\_reply.started":"2024-12-18T11:59:48.626524Z","shell.execute\_reply":"2024-12-18T11:59:49.237768Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:04:17.415758Z","iopub.execute\_input":"2024-12-18T12:04:17.416253Z","iopub.status.idle":"2024-12-18T12:04:17.424728Z","shell.execute\_reply.started":"2024-12-18T12:04:17.416211Z","shell.execute\_reply":"2024-12-18T12:04:17.423458Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:05:57.754087Z","iopub.execute\_input":"2024-12-18T12:05:57.755096Z","iopub.status.idle":"2024-12-18T12:05:58.239912Z","shell.execute\_reply.started":"2024-12-18T12:05:57.755042Z","shell.execute\_reply":"2024-12-18T12:05:58.238843Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:06:13.576413Z","iopub.execute\_input":"2024-12-18T12:06:13.576870Z","iopub.status.idle":"2024-12-18T12:06:14.229337Z","shell.execute\_reply.started":"2024-12-18T12:06:13.576833Z","shell.execute\_reply":"2024-12-18T12:06:14.228175Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:06:27.152538Z","iopub.execute\_input":"2024-12-18T12:06:27.153068Z","iopub.status.idle":"2024-12-18T12:06:27.850038Z","shell.execute\_reply.started":"2024-12-18T12:06:27.153024Z","shell.execute\_reply":"2024-12-18T12:06:27.846087Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:06:39.694593Z","iopub.execute\_input":"2024-12-18T12:06:39.695551Z","iopub.status.idle":"2024-12-18T12:06:42.290749Z","shell.execute\_reply.started":"2024-12-18T12:06:39.695487Z","shell.execute\_reply":"2024-12-18T12:06:42.289574Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:07:09.155886Z","iopub.execute\_input":"2024-12-18T12:07:09.156873Z","iopub.status.idle":"2024-12-18T12:07:09.342612Z","shell.execute\_reply.started":"2024-12-18T12:07:09.156828Z","shell.execute\_reply":"2024-12-18T12:07:09.340967Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:12:20.925586Z","iopub.execute\_input":"2024-12-18T12:12:20.926657Z","iopub.status.idle":"2024-12-18T12:12:21.627334Z","shell.execute\_reply.started":"2024-12-18T12:12:20.926608Z","shell.execute\_reply":"2024-12-18T12:12:21.626042Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

**NTP**

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:26:50.886000Z","iopub.execute\_input":"2024-12-18T12:26:50.886459Z","iopub.status.idle":"2024-12-18T12:27:13.410099Z","shell.execute\_reply.started":"2024-12-18T12:26:50.886384Z","shell.execute\_reply":"2024-12-18T12:27:13.409275Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-ntp/DrDoS\_NTP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:27:17.583193Z","iopub.execute\_input":"2024-12-18T12:27:17.583973Z","iopub.status.idle":"2024-12-18T12:27:17.589626Z","shell.execute\_reply.started":"2024-12-18T12:27:17.583932Z","shell.execute\_reply":"2024-12-18T12:27:17.588564Z"}}

# Check the column names

print("Columns in the dataset:", sampled\_df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:27:21.348342Z","iopub.execute\_input":"2024-12-18T12:27:21.348712Z","iopub.status.idle":"2024-12-18T12:27:21.576824Z","shell.execute\_reply.started":"2024-12-18T12:27:21.348682Z","shell.execute\_reply":"2024-12-18T12:27:21.575824Z"}}

# Step 3: Clean column names by stripping spaces

sampled\_df.columns = sampled\_df.columns.str.strip()

# Verify cleaned column names

print("Cleaned Columns:", sampled\_df.columns)

# Step 4: Convert 'Timestamp' column to datetime and sort the data

sampled\_df['Timestamp'] = pd.to\_datetime(sampled\_df['Timestamp'], errors='coerce')

sampled\_df = sampled\_df.dropna(subset=['Timestamp']) # Drop invalid timestamps

sampled\_df = sampled\_df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Verify the cleaned and sorted data

print("Data sorted by Timestamp:")

print(sampled\_df[['Timestamp']].head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:31:27.627009Z","iopub.execute\_input":"2024-12-18T12:31:27.627414Z","iopub.status.idle":"2024-12-18T12:31:27.666858Z","shell.execute\_reply.started":"2024-12-18T12:31:27.627357Z","shell.execute\_reply":"2024-12-18T12:31:27.665726Z"}}

from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Step 1: Select relevant features for modeling

features = [

'Flow Duration', 'Total Fwd Packets', 'Total Backward Packets',

'Total Length of Fwd Packets', 'Total Length of Bwd Packets',

'Flow Bytes/s', 'Flow Packets/s', 'Active Mean', 'Idle Mean'

]

X = sampled\_df[features]

# Step 2: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 3: Fill missing values with column means

X.fillna(X.mean(), inplace=True)

# Step 4: Normalize the features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:31:54.125593Z","iopub.execute\_input":"2024-12-18T12:31:54.125948Z","iopub.status.idle":"2024-12-18T12:32:07.113519Z","shell.execute\_reply.started":"2024-12-18T12:31:54.125919Z","shell.execute\_reply":"2024-12-18T12:32:07.112464Z"}}

import numpy as np

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dropout, Dense

# Step 1: Create Sequences for LSTM

def create\_sequences(data, time\_steps=10):

sequences, targets = [], []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

targets.append(data[i + time\_steps])

return np.array(sequences), np.array(targets)

# Time step for LSTM

time\_steps = 10

# Create sequences from the scaled data

X\_sequences, y\_sequences = create\_sequences(X\_scaled, time\_steps)

# Step 2: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_sequences, y\_sequences, test\_size=0.2, random\_state=42

)

print("Shape of training sequences:", X\_train.shape)

print("Shape of testing sequences:", X\_test.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:32:20.163033Z","iopub.execute\_input":"2024-12-18T12:32:20.163703Z","iopub.status.idle":"2024-12-18T12:36:00.742815Z","shell.execute\_reply.started":"2024-12-18T12:32:20.163664Z","shell.execute\_reply":"2024-12-18T12:36:00.741285Z"}}

# Step 3: Define the LSTM Model

model = Sequential([

LSTM(64, activation='tanh', input\_shape=(time\_steps, X\_train.shape[2])),

Dropout(0.2),

Dense(X\_train.shape[2], activation='linear') # Output layer matches feature size

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_test, y\_test),

epochs=20,

batch\_size=64,

verbose=1

)

# Step 5: Evaluate Reconstruction Errors

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data, verbose=0)

errors = np.mean(np.abs(data - predictions), axis=1) # Mean Absolute Error per sequence

return errors

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training data

threshold = np.percentile(train\_errors, 95) # 95th percentile

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_errors > threshold

print("Number of anomalies detected:", np.sum(test\_anomalies))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:36:31.084762Z","iopub.execute\_input":"2024-12-18T12:36:31.085138Z","iopub.status.idle":"2024-12-18T12:36:31.091737Z","shell.execute\_reply.started":"2024-12-18T12:36:31.085105Z","shell.execute\_reply":"2024-12-18T12:36:31.090459Z"}}

def calculate\_reconstruction\_error(data, model):

"""

Calculate reconstruction error for LSTM autoencoder.

:param data: Input data (3D: samples, timesteps, features)

:param model: Trained LSTM autoencoder

:return: Reconstruction errors (1D array)

"""

predictions = model.predict(data, verbose=0) # Shape: (samples, features)

# Use only the last timestep of the input sequences

data\_last\_step = data[:, -1, :] # Shape: (samples, features)

# Calculate Mean Absolute Error (MAE) per sequence

errors = np.mean(np.abs(data\_last\_step - predictions), axis=1)

return errors

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:36:46.292759Z","iopub.execute\_input":"2024-12-18T12:36:46.293141Z","iopub.status.idle":"2024-12-18T12:36:56.745266Z","shell.execute\_reply.started":"2024-12-18T12:36:46.293108Z","shell.execute\_reply":"2024-12-18T12:36:56.744223Z"}}

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training errors

threshold = np.percentile(train\_errors, 95) # e.g., 95th percentile

# Identify anomalies in test set

test\_anomalies = test\_errors > threshold

print(f"Anomaly detection threshold: {threshold}")

print(f"Number of anomalies detected: {np.sum(test\_anomalies)}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:37:06.618580Z","iopub.execute\_input":"2024-12-18T12:37:06.618952Z","iopub.status.idle":"2024-12-18T12:37:07.086439Z","shell.execute\_reply.started":"2024-12-18T12:37:06.618918Z","shell.execute\_reply":"2024-12-18T12:37:07.085266Z"}}

import matplotlib.pyplot as plt

plt.hist(train\_errors, bins=50, alpha=0.6, label='Train Errors')

plt.hist(test\_errors, bins=50, alpha=0.6, label='Test Errors')

plt.axvline(x=threshold, color='r', linestyle='--', label='Threshold')

plt.legend()

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.title('Reconstruction Error Distribution')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:38:06.509655Z","iopub.execute\_input":"2024-12-18T12:38:06.510425Z","iopub.status.idle":"2024-12-18T12:38:21.743024Z","shell.execute\_reply.started":"2024-12-18T12:38:06.510360Z","shell.execute\_reply":"2024-12-18T12:38:21.741998Z"}}

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-ntp/DrDoS\_NTP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df = df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column (Assume 'Label' contains attack type)

df['Label'] = df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 5: Separate Features and Target

X = df.drop(columns=['Label'])

y = df['Label']

print("Cleaned dataset shape:", X.shape, y.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:38:48.611498Z","iopub.execute\_input":"2024-12-18T12:38:48.611902Z","iopub.status.idle":"2024-12-18T12:39:11.885003Z","shell.execute\_reply.started":"2024-12-18T12:38:48.611866Z","shell.execute\_reply":"2024-12-18T12:39:11.883939Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-ntp/DrDoS\_NTP.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:39:28.618545Z","iopub.execute\_input":"2024-12-18T12:39:28.618940Z","iopub.status.idle":"2024-12-18T12:39:43.320782Z","shell.execute\_reply.started":"2024-12-18T12:39:28.618903Z","shell.execute\_reply":"2024-12-18T12:39:43.318965Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:40:32.537483Z","iopub.execute\_input":"2024-12-18T12:40:32.537862Z","iopub.status.idle":"2024-12-18T12:40:33.090343Z","shell.execute\_reply.started":"2024-12-18T12:40:32.537831Z","shell.execute\_reply":"2024-12-18T12:40:33.089325Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:40:52.665468Z","iopub.execute\_input":"2024-12-18T12:40:52.666149Z","iopub.status.idle":"2024-12-18T12:40:52.808748Z","shell.execute\_reply.started":"2024-12-18T12:40:52.666113Z","shell.execute\_reply":"2024-12-18T12:40:52.807582Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:41:27.653212Z","iopub.execute\_input":"2024-12-18T12:41:27.653616Z","iopub.status.idle":"2024-12-18T12:41:27.660573Z","shell.execute\_reply.started":"2024-12-18T12:41:27.653582Z","shell.execute\_reply":"2024-12-18T12:41:27.659466Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:41:43.757153Z","iopub.execute\_input":"2024-12-18T12:41:43.757558Z","iopub.status.idle":"2024-12-18T12:41:43.902941Z","shell.execute\_reply.started":"2024-12-18T12:41:43.757520Z","shell.execute\_reply":"2024-12-18T12:41:43.901890Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:41:58.933500Z","iopub.execute\_input":"2024-12-18T12:41:58.934210Z","iopub.status.idle":"2024-12-18T12:41:59.121143Z","shell.execute\_reply.started":"2024-12-18T12:41:58.934163Z","shell.execute\_reply":"2024-12-18T12:41:59.119982Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:42:10.778477Z","iopub.execute\_input":"2024-12-18T12:42:10.778870Z","iopub.status.idle":"2024-12-18T12:42:10.932000Z","shell.execute\_reply.started":"2024-12-18T12:42:10.778833Z","shell.execute\_reply":"2024-12-18T12:42:10.930918Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:42:24.163142Z","iopub.execute\_input":"2024-12-18T12:42:24.163572Z","iopub.status.idle":"2024-12-18T12:42:25.470093Z","shell.execute\_reply.started":"2024-12-18T12:42:24.163535Z","shell.execute\_reply":"2024-12-18T12:42:25.469123Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:42:38.834448Z","iopub.execute\_input":"2024-12-18T12:42:38.835430Z","iopub.status.idle":"2024-12-18T12:42:38.927055Z","shell.execute\_reply.started":"2024-12-18T12:42:38.835358Z","shell.execute\_reply":"2024-12-18T12:42:38.925948Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T12:42:55.700828Z","iopub.execute\_input":"2024-12-18T12:42:55.701237Z","iopub.status.idle":"2024-12-18T12:42:56.224443Z","shell.execute\_reply.started":"2024-12-18T12:42:55.701201Z","shell.execute\_reply":"2024-12-18T12:42:56.223337Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

**Python Codes SNMP**

# %% [code] {"jupyter":{"outputs\_hidden":false},"execution":{"iopub.status.busy":"2024-12-18T13:21:57.709671Z","iopub.execute\_input":"2024-12-18T13:21:57.710324Z","iopub.status.idle":"2024-12-18T13:23:05.680109Z","shell.execute\_reply.started":"2024-12-18T13:21:57.710278Z","shell.execute\_reply":"2024-12-18T13:23:05.679102Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-snmp/DrDoS\_SNMP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:23:10.236662Z","iopub.execute\_input":"2024-12-18T13:23:10.237890Z","iopub.status.idle":"2024-12-18T13:23:10.244469Z","shell.execute\_reply.started":"2024-12-18T13:23:10.237817Z","shell.execute\_reply":"2024-12-18T13:23:10.243285Z"}}

# Check the column names

print("Columns in the dataset:", sampled\_df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:23:13.699298Z","iopub.execute\_input":"2024-12-18T13:23:13.699684Z","iopub.status.idle":"2024-12-18T13:23:14.950264Z","shell.execute\_reply.started":"2024-12-18T13:23:13.699648Z","shell.execute\_reply":"2024-12-18T13:23:14.948991Z"}}

# Step 3: Clean column names by stripping spaces

sampled\_df.columns = sampled\_df.columns.str.strip()

# Verify cleaned column names

print("Cleaned Columns:", sampled\_df.columns)

# Step 4: Convert 'Timestamp' column to datetime and sort the data

sampled\_df['Timestamp'] = pd.to\_datetime(sampled\_df['Timestamp'], errors='coerce')

sampled\_df = sampled\_df.dropna(subset=['Timestamp']) # Drop invalid timestamps

sampled\_df = sampled\_df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Verify the cleaned and sorted data

print("Data sorted by Timestamp:")

print(sampled\_df[['Timestamp']].head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:23:18.050802Z","iopub.execute\_input":"2024-12-18T13:23:18.051250Z","iopub.status.idle":"2024-12-18T13:23:18.210395Z","shell.execute\_reply.started":"2024-12-18T13:23:18.051215Z","shell.execute\_reply":"2024-12-18T13:23:18.209181Z"}}

from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Step 1: Select relevant features for modeling

features = [

'Flow Duration', 'Total Fwd Packets', 'Total Backward Packets',

'Total Length of Fwd Packets', 'Total Length of Bwd Packets',

'Flow Bytes/s', 'Flow Packets/s', 'Active Mean', 'Idle Mean'

]

X = sampled\_df[features]

# Step 2: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 3: Fill missing values with column means

X.fillna(X.mean(), inplace=True)

# Step 4: Normalize the features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:23:21.849376Z","iopub.execute\_input":"2024-12-18T13:23:21.849879Z","iopub.status.idle":"2024-12-18T13:23:36.670915Z","shell.execute\_reply.started":"2024-12-18T13:23:21.849833Z","shell.execute\_reply":"2024-12-18T13:23:36.669432Z"}}

import numpy as np

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dropout, Dense

# Step 1: Create Sequences for LSTM

def create\_sequences(data, time\_steps=10):

sequences, targets = [], []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

targets.append(data[i + time\_steps])

return np.array(sequences), np.array(targets)

# Time step for LSTM

time\_steps = 10

# Create sequences from the scaled data

X\_sequences, y\_sequences = create\_sequences(X\_scaled, time\_steps)

# Step 2: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_sequences, y\_sequences, test\_size=0.2, random\_state=42

)

print("Shape of training sequences:", X\_train.shape)

print("Shape of testing sequences:", X\_test.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:24:08.704102Z","iopub.execute\_input":"2024-12-18T13:24:08.704979Z","iopub.status.idle":"2024-12-18T13:41:05.592475Z","shell.execute\_reply.started":"2024-12-18T13:24:08.704932Z","shell.execute\_reply":"2024-12-18T13:41:05.590656Z"}}

# Step 3: Define the LSTM Model

model = Sequential([

LSTM(64, activation='tanh', input\_shape=(time\_steps, X\_train.shape[2])),

Dropout(0.2),

Dense(X\_train.shape[2], activation='linear') # Output layer matches feature size

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_test, y\_test),

epochs=20,

batch\_size=64,

verbose=1

)

# Step 5: Evaluate Reconstruction Errors

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data, verbose=0)

errors = np.mean(np.abs(data - predictions), axis=1) # Mean Absolute Error per sequence

return errors

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training data

threshold = np.percentile(train\_errors, 95) # 95th percentile

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_errors > threshold

print("Number of anomalies detected:", np.sum(test\_anomalies))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:43:51.444135Z","iopub.execute\_input":"2024-12-18T13:43:51.444651Z","iopub.status.idle":"2024-12-18T13:43:51.452064Z","shell.execute\_reply.started":"2024-12-18T13:43:51.444604Z","shell.execute\_reply":"2024-12-18T13:43:51.450607Z"}}

def calculate\_reconstruction\_error(data, model):

"""

Calculate reconstruction error for LSTM autoencoder.

:param data: Input data (3D: samples, timesteps, features)

:param model: Trained LSTM autoencoder

:return: Reconstruction errors (1D array)

"""

predictions = model.predict(data, verbose=0) # Shape: (samples, features)

# Use only the last timestep of the input sequences

data\_last\_step = data[:, -1, :] # Shape: (samples, features)

# Calculate Mean Absolute Error (MAE) per sequence

errors = np.mean(np.abs(data\_last\_step - predictions), axis=1)

return errors

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:43:57.098126Z","iopub.execute\_input":"2024-12-18T13:43:57.098604Z","iopub.status.idle":"2024-12-18T13:44:47.544008Z","shell.execute\_reply.started":"2024-12-18T13:43:57.098563Z","shell.execute\_reply":"2024-12-18T13:44:47.542867Z"}}

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training errors

threshold = np.percentile(train\_errors, 95) # e.g., 95th percentile

# Identify anomalies in test set

test\_anomalies = test\_errors > threshold

print(f"Anomaly detection threshold: {threshold}")

print(f"Number of anomalies detected: {np.sum(test\_anomalies)}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:44:55.840715Z","iopub.execute\_input":"2024-12-18T13:44:55.841295Z","iopub.status.idle":"2024-12-18T13:44:56.369424Z","shell.execute\_reply.started":"2024-12-18T13:44:55.841250Z","shell.execute\_reply":"2024-12-18T13:44:56.368327Z"}}

import matplotlib.pyplot as plt

plt.hist(train\_errors, bins=50, alpha=0.6, label='Train Errors')

plt.hist(test\_errors, bins=50, alpha=0.6, label='Test Errors')

plt.axvline(x=threshold, color='r', linestyle='--', label='Threshold')

plt.legend()

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.title('Reconstruction Error Distribution')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:46:07.866610Z","iopub.execute\_input":"2024-12-18T13:46:07.867200Z","iopub.status.idle":"2024-12-18T13:47:14.055095Z","shell.execute\_reply.started":"2024-12-18T13:46:07.867154Z","shell.execute\_reply":"2024-12-18T13:47:14.053009Z"}}

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-snmp/DrDoS\_SNMP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df = df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column (Assume 'Label' contains attack type)

df['Label'] = df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 5: Separate Features and Target

X = df.drop(columns=['Label'])

y = df['Label']

print("Cleaned dataset shape:", X.shape, y.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:47:26.407957Z","iopub.execute\_input":"2024-12-18T13:47:26.408587Z","iopub.status.idle":"2024-12-18T13:48:58.315922Z","shell.execute\_reply.started":"2024-12-18T13:47:26.408538Z","shell.execute\_reply":"2024-12-18T13:48:58.314345Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-snmp/DrDoS\_SNMP.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:49:20.710725Z","iopub.execute\_input":"2024-12-18T13:49:20.711270Z","iopub.status.idle":"2024-12-18T13:50:44.637715Z","shell.execute\_reply.started":"2024-12-18T13:49:20.711224Z","shell.execute\_reply":"2024-12-18T13:50:44.636610Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:51:34.441734Z","iopub.execute\_input":"2024-12-18T13:51:34.442500Z","iopub.status.idle":"2024-12-18T13:51:35.059252Z","shell.execute\_reply.started":"2024-12-18T13:51:34.442451Z","shell.execute\_reply":"2024-12-18T13:51:35.058044Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:51:43.495708Z","iopub.execute\_input":"2024-12-18T13:51:43.496627Z","iopub.status.idle":"2024-12-18T13:51:44.098928Z","shell.execute\_reply.started":"2024-12-18T13:51:43.496582Z","shell.execute\_reply":"2024-12-18T13:51:44.097815Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:51:51.515146Z","iopub.execute\_input":"2024-12-18T13:51:51.515603Z","iopub.status.idle":"2024-12-18T13:51:51.523968Z","shell.execute\_reply.started":"2024-12-18T13:51:51.515563Z","shell.execute\_reply":"2024-12-18T13:51:51.522521Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:51:58.579835Z","iopub.execute\_input":"2024-12-18T13:51:58.580303Z","iopub.status.idle":"2024-12-18T13:51:59.344259Z","shell.execute\_reply.started":"2024-12-18T13:51:58.580265Z","shell.execute\_reply":"2024-12-18T13:51:59.342702Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:52:08.008926Z","iopub.execute\_input":"2024-12-18T13:52:08.009420Z","iopub.status.idle":"2024-12-18T13:52:08.899317Z","shell.execute\_reply.started":"2024-12-18T13:52:08.009378Z","shell.execute\_reply":"2024-12-18T13:52:08.898153Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:52:16.956859Z","iopub.execute\_input":"2024-12-18T13:52:16.958037Z","iopub.status.idle":"2024-12-18T13:52:17.638649Z","shell.execute\_reply.started":"2024-12-18T13:52:16.957977Z","shell.execute\_reply":"2024-12-18T13:52:17.637433Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:52:24.766135Z","iopub.execute\_input":"2024-12-18T13:52:24.766623Z","iopub.status.idle":"2024-12-18T13:52:28.089245Z","shell.execute\_reply.started":"2024-12-18T13:52:24.766581Z","shell.execute\_reply":"2024-12-18T13:52:28.087853Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:52:35.659448Z","iopub.execute\_input":"2024-12-18T13:52:35.659995Z","iopub.status.idle":"2024-12-18T13:52:35.909078Z","shell.execute\_reply.started":"2024-12-18T13:52:35.659951Z","shell.execute\_reply":"2024-12-18T13:52:35.907606Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T13:52:43.314409Z","iopub.execute\_input":"2024-12-18T13:52:43.314958Z","iopub.status.idle":"2024-12-18T13:52:44.030362Z","shell.execute\_reply.started":"2024-12-18T13:52:43.314906Z","shell.execute\_reply":"2024-12-18T13:52:44.029091Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

**Python Codes SSDP**

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:11:28.976090Z","iopub.execute\_input":"2024-12-18T14:11:28.976479Z","iopub.status.idle":"2024-12-18T14:12:19.242888Z","shell.execute\_reply.started":"2024-12-18T14:11:28.976442Z","shell.execute\_reply":"2024-12-18T14:12:19.241469Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-ssdp/DrDoS\_SSDP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:15:01.688842Z","iopub.execute\_input":"2024-12-18T14:15:01.689991Z","iopub.status.idle":"2024-12-18T14:15:01.700439Z","shell.execute\_reply.started":"2024-12-18T14:15:01.689942Z","shell.execute\_reply":"2024-12-18T14:15:01.698999Z"}}

# Check the column names

print("Columns in the dataset:", sampled\_df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:15:05.308202Z","iopub.execute\_input":"2024-12-18T14:15:05.308950Z","iopub.status.idle":"2024-12-18T14:15:05.924809Z","shell.execute\_reply.started":"2024-12-18T14:15:05.308884Z","shell.execute\_reply":"2024-12-18T14:15:05.923308Z"}}

# Step 3: Clean column names by stripping spaces

sampled\_df.columns = sampled\_df.columns.str.strip()

# Verify cleaned column names

print("Cleaned Columns:", sampled\_df.columns)

# Step 4: Convert 'Timestamp' column to datetime and sort the data

sampled\_df['Timestamp'] = pd.to\_datetime(sampled\_df['Timestamp'], errors='coerce')

sampled\_df = sampled\_df.dropna(subset=['Timestamp']) # Drop invalid timestamps

sampled\_df = sampled\_df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Verify the cleaned and sorted data

print("Data sorted by Timestamp:")

print(sampled\_df[['Timestamp']].head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:15:09.713802Z","iopub.execute\_input":"2024-12-18T14:15:09.714282Z","iopub.status.idle":"2024-12-18T14:15:09.805251Z","shell.execute\_reply.started":"2024-12-18T14:15:09.714244Z","shell.execute\_reply":"2024-12-18T14:15:09.803769Z"}}

from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Step 1: Select relevant features for modeling

features = [

'Flow Duration', 'Total Fwd Packets', 'Total Backward Packets',

'Total Length of Fwd Packets', 'Total Length of Bwd Packets',

'Flow Bytes/s', 'Flow Packets/s', 'Active Mean', 'Idle Mean'

]

X = sampled\_df[features]

# Step 2: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 3: Fill missing values with column means

X.fillna(X.mean(), inplace=True)

# Step 4: Normalize the features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:15:13.847036Z","iopub.execute\_input":"2024-12-18T14:15:13.847412Z","iopub.status.idle":"2024-12-18T14:15:29.413525Z","shell.execute\_reply.started":"2024-12-18T14:15:13.847380Z","shell.execute\_reply":"2024-12-18T14:15:29.412173Z"}}

import numpy as np

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dropout, Dense

# Step 1: Create Sequences for LSTM

def create\_sequences(data, time\_steps=10):

sequences, targets = [], []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

targets.append(data[i + time\_steps])

return np.array(sequences), np.array(targets)

# Time step for LSTM

time\_steps = 10

# Create sequences from the scaled data

X\_sequences, y\_sequences = create\_sequences(X\_scaled, time\_steps)

# Step 2: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_sequences, y\_sequences, test\_size=0.2, random\_state=42

)

print("Shape of training sequences:", X\_train.shape)

print("Shape of testing sequences:", X\_test.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:15:38.429147Z","iopub.execute\_input":"2024-12-18T14:15:38.430041Z","iopub.status.idle":"2024-12-18T14:27:33.840333Z","shell.execute\_reply.started":"2024-12-18T14:15:38.429992Z","shell.execute\_reply":"2024-12-18T14:27:33.837923Z"}}

# Step 3: Define the LSTM Model

model = Sequential([

LSTM(64, activation='tanh', input\_shape=(time\_steps, X\_train.shape[2])),

Dropout(0.2),

Dense(X\_train.shape[2], activation='linear') # Output layer matches feature size

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_test, y\_test),

epochs=20,

batch\_size=64,

verbose=1

)

# Step 5: Evaluate Reconstruction Errors

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data, verbose=0)

errors = np.mean(np.abs(data - predictions), axis=1) # Mean Absolute Error per sequence

return errors

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training data

threshold = np.percentile(train\_errors, 95) # 95th percentile

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_errors > threshold

print("Number of anomalies detected:", np.sum(test\_anomalies))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:40:22.601842Z","iopub.execute\_input":"2024-12-18T14:40:22.603657Z","iopub.status.idle":"2024-12-18T14:40:22.611594Z","shell.execute\_reply.started":"2024-12-18T14:40:22.603581Z","shell.execute\_reply":"2024-12-18T14:40:22.610260Z"}}

def calculate\_reconstruction\_error(data, model):

"""

Calculate reconstruction error for LSTM autoencoder.

:param data: Input data (3D: samples, timesteps, features)

:param model: Trained LSTM autoencoder

:return: Reconstruction errors (1D array)

"""

predictions = model.predict(data, verbose=0) # Shape: (samples, features)

# Use only the last timestep of the input sequences

data\_last\_step = data[:, -1, :] # Shape: (samples, features)

# Calculate Mean Absolute Error (MAE) per sequence

errors = np.mean(np.abs(data\_last\_step - predictions), axis=1)

return errors

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:40:25.675676Z","iopub.execute\_input":"2024-12-18T14:40:25.676675Z","iopub.status.idle":"2024-12-18T14:40:58.601840Z","shell.execute\_reply.started":"2024-12-18T14:40:25.676633Z","shell.execute\_reply":"2024-12-18T14:40:58.600262Z"}}

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training errors

threshold = np.percentile(train\_errors, 95) # e.g., 95th percentile

# Identify anomalies in test set

test\_anomalies = test\_errors > threshold

print(f"Anomaly detection threshold: {threshold}")

print(f"Number of anomalies detected: {np.sum(test\_anomalies)}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:41:09.614469Z","iopub.execute\_input":"2024-12-18T14:41:09.614940Z","iopub.status.idle":"2024-12-18T14:41:10.129370Z","shell.execute\_reply.started":"2024-12-18T14:41:09.614902Z","shell.execute\_reply":"2024-12-18T14:41:10.128263Z"}}

import matplotlib.pyplot as plt

plt.hist(train\_errors, bins=50, alpha=0.6, label='Train Errors')

plt.hist(test\_errors, bins=50, alpha=0.6, label='Test Errors')

plt.axvline(x=threshold, color='r', linestyle='--', label='Threshold')

plt.legend()

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.title('Reconstruction Error Distribution')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:41:15.639943Z","iopub.execute\_input":"2024-12-18T14:41:15.640337Z","iopub.status.idle":"2024-12-18T14:41:51.286970Z","shell.execute\_reply.started":"2024-12-18T14:41:15.640303Z","shell.execute\_reply":"2024-12-18T14:41:51.285437Z"}}

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-ssdp/DrDoS\_SSDP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df = df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column (Assume 'Label' contains attack type)

df['Label'] = df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 5: Separate Features and Target

X = df.drop(columns=['Label'])

y = df['Label']

print("Cleaned dataset shape:", X.shape, y.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:43:42.780505Z","iopub.execute\_input":"2024-12-18T14:43:42.781046Z","iopub.status.idle":"2024-12-18T14:44:53.514291Z","shell.execute\_reply.started":"2024-12-18T14:43:42.781006Z","shell.execute\_reply":"2024-12-18T14:44:53.512803Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-ssdp/DrDoS\_SSDP.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:44:58.683737Z","iopub.execute\_input":"2024-12-18T14:44:58.684201Z","iopub.status.idle":"2024-12-18T14:46:03.742800Z","shell.execute\_reply.started":"2024-12-18T14:44:58.684135Z","shell.execute\_reply":"2024-12-18T14:46:03.741579Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:46:31.630075Z","iopub.execute\_input":"2024-12-18T14:46:31.630525Z","iopub.status.idle":"2024-12-18T14:46:32.815173Z","shell.execute\_reply.started":"2024-12-18T14:46:31.630490Z","shell.execute\_reply":"2024-12-18T14:46:32.813768Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:46:45.033260Z","iopub.execute\_input":"2024-12-18T14:46:45.034119Z","iopub.status.idle":"2024-12-18T14:46:45.350851Z","shell.execute\_reply.started":"2024-12-18T14:46:45.034078Z","shell.execute\_reply":"2024-12-18T14:46:45.347556Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:46:52.581169Z","iopub.execute\_input":"2024-12-18T14:46:52.581810Z","iopub.status.idle":"2024-12-18T14:46:52.592305Z","shell.execute\_reply.started":"2024-12-18T14:46:52.581751Z","shell.execute\_reply":"2024-12-18T14:46:52.590656Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:46:59.875470Z","iopub.execute\_input":"2024-12-18T14:46:59.875959Z","iopub.status.idle":"2024-12-18T14:47:00.140998Z","shell.execute\_reply.started":"2024-12-18T14:46:59.875922Z","shell.execute\_reply":"2024-12-18T14:47:00.139906Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:47:06.266337Z","iopub.execute\_input":"2024-12-18T14:47:06.267959Z","iopub.status.idle":"2024-12-18T14:47:06.594654Z","shell.execute\_reply.started":"2024-12-18T14:47:06.267884Z","shell.execute\_reply":"2024-12-18T14:47:06.593233Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:47:14.868538Z","iopub.execute\_input":"2024-12-18T14:47:14.869698Z","iopub.status.idle":"2024-12-18T14:47:15.750025Z","shell.execute\_reply.started":"2024-12-18T14:47:14.869653Z","shell.execute\_reply":"2024-12-18T14:47:15.748493Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:47:24.338914Z","iopub.execute\_input":"2024-12-18T14:47:24.339374Z","iopub.status.idle":"2024-12-18T14:47:26.473763Z","shell.execute\_reply.started":"2024-12-18T14:47:24.339336Z","shell.execute\_reply":"2024-12-18T14:47:26.472960Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:47:33.393150Z","iopub.execute\_input":"2024-12-18T14:47:33.393576Z","iopub.status.idle":"2024-12-18T14:47:33.524316Z","shell.execute\_reply.started":"2024-12-18T14:47:33.393540Z","shell.execute\_reply":"2024-12-18T14:47:33.522808Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T14:47:40.773286Z","iopub.execute\_input":"2024-12-18T14:47:40.773802Z","iopub.status.idle":"2024-12-18T14:47:41.435457Z","shell.execute\_reply.started":"2024-12-18T14:47:40.773759Z","shell.execute\_reply":"2024-12-18T14:47:41.434060Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)

**Python Codes UDP**

# %% [code] {"jupyter":{"outputs\_hidden":false},"execution":{"iopub.status.busy":"2024-12-18T15:29:09.319385Z","iopub.execute\_input":"2024-12-18T15:29:09.319930Z","iopub.status.idle":"2024-12-18T15:30:02.880147Z","shell.execute\_reply.started":"2024-12-18T15:29:09.319849Z","shell.execute\_reply":"2024-12-18T15:30:02.878854Z"}}

# Import necessary libraries

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

# Step 1: Load the dataset

file\_path = "/kaggle/input/drdos-udp/DrDoS\_UDP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Apply random sampling to reduce the dataset size

sampled\_df = df.sample(frac=0.1, random\_state=42) # Adjust 'frac' for percentage of data (10% in this case)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:30:30.347083Z","iopub.execute\_input":"2024-12-18T15:30:30.347470Z","iopub.status.idle":"2024-12-18T15:30:30.354669Z","shell.execute\_reply.started":"2024-12-18T15:30:30.347435Z","shell.execute\_reply":"2024-12-18T15:30:30.353287Z"}}

# Check the column names

print("Columns in the dataset:", sampled\_df.columns)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:30:33.673189Z","iopub.execute\_input":"2024-12-18T15:30:33.673597Z","iopub.status.idle":"2024-12-18T15:30:34.377216Z","shell.execute\_reply.started":"2024-12-18T15:30:33.673562Z","shell.execute\_reply":"2024-12-18T15:30:34.376017Z"}}

# Step 3: Clean column names by stripping spaces

sampled\_df.columns = sampled\_df.columns.str.strip()

# Verify cleaned column names

print("Cleaned Columns:", sampled\_df.columns)

# Step 4: Convert 'Timestamp' column to datetime and sort the data

sampled\_df['Timestamp'] = pd.to\_datetime(sampled\_df['Timestamp'], errors='coerce')

sampled\_df = sampled\_df.dropna(subset=['Timestamp']) # Drop invalid timestamps

sampled\_df = sampled\_df.sort\_values(by='Timestamp').reset\_index(drop=True)

# Verify the cleaned and sorted data

print("Data sorted by Timestamp:")

print(sampled\_df[['Timestamp']].head())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:30:40.177087Z","iopub.execute\_input":"2024-12-18T15:30:40.177617Z","iopub.status.idle":"2024-12-18T15:30:40.268756Z","shell.execute\_reply.started":"2024-12-18T15:30:40.177563Z","shell.execute\_reply":"2024-12-18T15:30:40.267710Z"}}

from sklearn.preprocessing import MinMaxScaler

import numpy as np

# Step 1: Select relevant features for modeling

features = [

'Flow Duration', 'Total Fwd Packets', 'Total Backward Packets',

'Total Length of Fwd Packets', 'Total Length of Bwd Packets',

'Flow Bytes/s', 'Flow Packets/s', 'Active Mean', 'Idle Mean'

]

X = sampled\_df[features]

# Step 2: Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Step 3: Fill missing values with column means

X.fillna(X.mean(), inplace=True)

# Step 4: Normalize the features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# Verify scaling

print("Shape of scaled data:", X\_scaled.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:30:44.176777Z","iopub.execute\_input":"2024-12-18T15:30:44.177231Z","iopub.status.idle":"2024-12-18T15:30:59.413045Z","shell.execute\_reply.started":"2024-12-18T15:30:44.177192Z","shell.execute\_reply":"2024-12-18T15:30:59.411839Z"}}

import numpy as np

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dropout, Dense

# Step 1: Create Sequences for LSTM

def create\_sequences(data, time\_steps=10):

sequences, targets = [], []

for i in range(len(data) - time\_steps):

sequences.append(data[i:i + time\_steps])

targets.append(data[i + time\_steps])

return np.array(sequences), np.array(targets)

# Time step for LSTM

time\_steps = 10

# Create sequences from the scaled data

X\_sequences, y\_sequences = create\_sequences(X\_scaled, time\_steps)

# Step 2: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_sequences, y\_sequences, test\_size=0.2, random\_state=42

)

print("Shape of training sequences:", X\_train.shape)

print("Shape of testing sequences:", X\_test.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:31:09.239074Z","iopub.execute\_input":"2024-12-18T15:31:09.239821Z","iopub.status.idle":"2024-12-18T15:42:52.980195Z","shell.execute\_reply.started":"2024-12-18T15:31:09.239779Z","shell.execute\_reply":"2024-12-18T15:42:52.977990Z"}}

# Step 3: Define the LSTM Model

model = Sequential([

LSTM(64, activation='tanh', input\_shape=(time\_steps, X\_train.shape[2])),

Dropout(0.2),

Dense(X\_train.shape[2], activation='linear') # Output layer matches feature size

])

model.compile(optimizer='adam', loss='mse')

model.summary()

# Step 4: Train the Model

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_test, y\_test),

epochs=20,

batch\_size=64,

verbose=1

)

# Step 5: Evaluate Reconstruction Errors

def calculate\_reconstruction\_error(data, model):

predictions = model.predict(data, verbose=0)

errors = np.mean(np.abs(data - predictions), axis=1) # Mean Absolute Error per sequence

return errors

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training data

threshold = np.percentile(train\_errors, 95) # 95th percentile

print("Reconstruction error threshold:", threshold)

# Detect anomalies

test\_anomalies = test\_errors > threshold

print("Number of anomalies detected:", np.sum(test\_anomalies))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:43:08.450898Z","iopub.execute\_input":"2024-12-18T15:43:08.451364Z","iopub.status.idle":"2024-12-18T15:43:08.458733Z","shell.execute\_reply.started":"2024-12-18T15:43:08.451324Z","shell.execute\_reply":"2024-12-18T15:43:08.457564Z"}}

def calculate\_reconstruction\_error(data, model):

"""

Calculate reconstruction error for LSTM autoencoder.

:param data: Input data (3D: samples, timesteps, features)

:param model: Trained LSTM autoencoder

:return: Reconstruction errors (1D array)

"""

predictions = model.predict(data, verbose=0) # Shape: (samples, features)

# Use only the last timestep of the input sequences

data\_last\_step = data[:, -1, :] # Shape: (samples, features)

# Calculate Mean Absolute Error (MAE) per sequence

errors = np.mean(np.abs(data\_last\_step - predictions), axis=1)

return errors

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:46:30.934020Z","iopub.execute\_input":"2024-12-18T15:46:30.934551Z","iopub.status.idle":"2024-12-18T15:47:05.802311Z","shell.execute\_reply.started":"2024-12-18T15:46:30.934510Z","shell.execute\_reply":"2024-12-18T15:47:05.800971Z"}}

# Calculate reconstruction errors on training and test sets

train\_errors = calculate\_reconstruction\_error(X\_train, model)

test\_errors = calculate\_reconstruction\_error(X\_test, model)

# Set anomaly detection threshold based on training errors

threshold = np.percentile(train\_errors, 95) # e.g., 95th percentile

# Identify anomalies in test set

test\_anomalies = test\_errors > threshold

print(f"Anomaly detection threshold: {threshold}")

print(f"Number of anomalies detected: {np.sum(test\_anomalies)}")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:47:13.823059Z","iopub.execute\_input":"2024-12-18T15:47:13.823477Z","iopub.status.idle":"2024-12-18T15:47:14.345634Z","shell.execute\_reply.started":"2024-12-18T15:47:13.823444Z","shell.execute\_reply":"2024-12-18T15:47:14.344454Z"}}

import matplotlib.pyplot as plt

plt.hist(train\_errors, bins=50, alpha=0.6, label='Train Errors')

plt.hist(test\_errors, bins=50, alpha=0.6, label='Test Errors')

plt.axvline(x=threshold, color='r', linestyle='--', label='Threshold')

plt.legend()

plt.xlabel('Reconstruction Error')

plt.ylabel('Frequency')

plt.title('Reconstruction Error Distribution')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:47:20.222138Z","iopub.execute\_input":"2024-12-18T15:47:20.222563Z","iopub.status.idle":"2024-12-18T15:47:59.012426Z","shell.execute\_reply.started":"2024-12-18T15:47:20.222525Z","shell.execute\_reply":"2024-12-18T15:47:59.011062Z"}}

# Step 1: Load Data

file\_path = "/kaggle/input/drdos-udp/DrDoS\_UDP.csv"

df = pd.read\_csv(file\_path)

# Step 2: Clean Column Names

df.columns = df.columns.str.strip() # Remove leading/trailing spaces from column names

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

df = df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column (Assume 'Label' contains attack type)

df['Label'] = df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0) # 1: Attack, 0: Normal

# Step 5: Separate Features and Target

X = df.drop(columns=['Label'])

y = df['Label']

print("Cleaned dataset shape:", X.shape, y.shape)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:49:04.787182Z","iopub.execute\_input":"2024-12-18T15:49:04.787624Z","iopub.status.idle":"2024-12-18T15:50:06.313478Z","shell.execute\_reply.started":"2024-12-18T15:49:04.787577Z","shell.execute\_reply":"2024-12-18T15:50:06.312264Z"}}

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Step 1: Load and Sample Data

file\_path = "/kaggle/input/drdos-udp/DrDoS\_UDP.csv"

df = pd.read\_csv(file\_path, low\_memory=False)

df.columns = df.columns.str.strip() # Clean column names

# Step 2: Random Sampling

sampled\_df = df.sample(frac=0.1, random\_state=42)

# Step 3: Drop Unnecessary Columns

columns\_to\_drop = ['Unnamed: 0', 'Flow ID', 'Source IP', 'Destination IP', 'Timestamp']

sampled\_df = sampled\_df.drop(columns=columns\_to\_drop)

# Step 4: Encode Target Column ('Label': 1 for Attack, 0 for BENIGN)

sampled\_df['Label'] = sampled\_df['Label'].apply(lambda x: 1 if x != 'BENIGN' else 0)

# Step 5: Separate Features and Target

X = sampled\_df.drop(columns=['Label'])

y = sampled\_df['Label']

# Step 6: Identify Categorical and Numerical Columns

categorical\_cols = X.select\_dtypes(include=['object']).columns.tolist()

numerical\_cols = X.select\_dtypes(include=['int64', 'float64']).columns.tolist()

print("Categorical Columns:", categorical\_cols)

print("Numerical Columns:", numerical\_cols)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:50:13.623111Z","iopub.execute\_input":"2024-12-18T15:50:13.623731Z","iopub.status.idle":"2024-12-18T15:51:15.830907Z","shell.execute\_reply.started":"2024-12-18T15:50:13.623664Z","shell.execute\_reply":"2024-12-18T15:51:15.829620Z"}}

# Step 7: Handle Infinite, NaN Values, and Invalid Data

# Convert all columns to numeric, replacing invalid entries with NaN

for col in numerical\_cols:

X[col] = pd.to\_numeric(X[col], errors='coerce')

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Drop rows with NaN values in X and keep y in sync

valid\_indices = X.dropna().index

X = X.loc[valid\_indices]

y = y.loc[valid\_indices]

# Verify there are no NaN values left

print("Remaining NaN Values:", X.isna().sum().sum())

# Step 8: Feature Transformation (Scaling and Encoding)

preprocessor = ColumnTransformer(

transformers=[

('num', StandardScaler(), numerical\_cols), # Scale numerical columns

('cat', OneHotEncoder(handle\_unknown='ignore'), categorical\_cols) # Encode categorical columns

]

)

X\_transformed = preprocessor.fit\_transform(X)

# Step 9: Split Data into Training and Testing Sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_transformed, y, test\_size=0.2, random\_state=42, stratify=y

)

# Step 10: Train Logistic Regression Model

log\_reg = LogisticRegression(max\_iter=1000, random\_state=42)

log\_reg.fit(X\_train, y\_train)

# Step 11: Make Predictions

y\_pred = log\_reg.predict(X\_test)

# Step 12: Evaluate the Model

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:29.342605Z","iopub.execute\_input":"2024-12-18T15:51:29.343401Z","iopub.status.idle":"2024-12-18T15:51:29.960987Z","shell.execute\_reply.started":"2024-12-18T15:51:29.343357Z","shell.execute\_reply":"2024-12-18T15:51:29.959774Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Normal', 'Attack'], yticklabels=['Normal', 'Attack'])

plt.title('Confusion Matrix')

plt.xlabel('Predicted Labels')

plt.ylabel('True Labels')

plt.show()

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:33.634478Z","iopub.execute\_input":"2024-12-18T15:51:33.635435Z","iopub.status.idle":"2024-12-18T15:51:34.026262Z","shell.execute\_reply.started":"2024-12-18T15:51:33.635373Z","shell.execute\_reply":"2024-12-18T15:51:34.024952Z"}}

# Step 7: Handle Infinite, NaN, and Non-Numeric Values

# Replace infinite values with NaN

X.replace([np.inf, -np.inf], np.nan, inplace=True)

# Identify and drop non-numeric columns

non\_numeric\_columns = X.select\_dtypes(include=['object']).columns

print("Non-numeric columns detected:", non\_numeric\_columns)

# Option 1: Drop non-numeric columns if irrelevant

X = X.drop(columns=non\_numeric\_columns)

# Option 2: If the non-numeric columns are essential, convert them to numeric (if possible)

# Uncomment the following line if you want to try conversion

# X[non\_numeric\_columns] = X[non\_numeric\_columns].apply(pd.to\_numeric, errors='coerce')

# Fill NaN values with column means

X = X.fillna(X.mean())

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:37.657510Z","iopub.execute\_input":"2024-12-18T15:51:37.658089Z","iopub.status.idle":"2024-12-18T15:51:37.667698Z","shell.execute\_reply.started":"2024-12-18T15:51:37.658036Z","shell.execute\_reply":"2024-12-18T15:51:37.666240Z"}}

# Ensure column names are stripped of extra spaces

X.columns = X.columns.str.strip()

# Check if SimillarHTTP exists, then handle it

if 'SimillarHTTP' in X.columns:

# Option 1: Drop the column

X = X.drop(columns=['SimillarHTTP'])

print("Dropped 'SimillarHTTP' column.")

# Option 2 (if relevant): Encode the column

# Apply one-hot encoding or label encoding as needed

else:

print("'SimillarHTTP' column not found in X. Skipping.")

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:41.268336Z","iopub.execute\_input":"2024-12-18T15:51:41.268763Z","iopub.status.idle":"2024-12-18T15:51:41.626078Z","shell.execute\_reply.started":"2024-12-18T15:51:41.268723Z","shell.execute\_reply":"2024-12-18T15:51:41.624976Z"}}

X.replace([np.inf, -np.inf], np.nan, inplace=True)

X.fillna(X.mean(), inplace=True)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:45.247888Z","iopub.execute\_input":"2024-12-18T15:51:45.248302Z","iopub.status.idle":"2024-12-18T15:51:45.727928Z","shell.execute\_reply.started":"2024-12-18T15:51:45.248255Z","shell.execute\_reply":"2024-12-18T15:51:45.726533Z"}}

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

# Scale numerical features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:50.012033Z","iopub.execute\_input":"2024-12-18T15:51:50.013173Z","iopub.status.idle":"2024-12-18T15:51:50.418662Z","shell.execute\_reply.started":"2024-12-18T15:51:50.013124Z","shell.execute\_reply":"2024-12-18T15:51:50.417327Z"}}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:53.679798Z","iopub.execute\_input":"2024-12-18T15:51:53.680289Z","iopub.status.idle":"2024-12-18T15:51:56.077089Z","shell.execute\_reply.started":"2024-12-18T15:51:53.680250Z","shell.execute\_reply":"2024-12-18T15:51:56.074653Z"}}

from xgboost import XGBClassifier

xgb\_model = XGBClassifier(

max\_depth=6,

learning\_rate=0.1,

n\_estimators=100,

verbosity=1,

random\_state=42

)

xgb\_model.fit(X\_train, y\_train)

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:51:59.568384Z","iopub.execute\_input":"2024-12-18T15:51:59.568808Z","iopub.status.idle":"2024-12-18T15:51:59.710267Z","shell.execute\_reply.started":"2024-12-18T15:51:59.568770Z","shell.execute\_reply":"2024-12-18T15:51:59.709080Z"}}

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

y\_pred = xgb\_model.predict(X\_test)

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

print("\nAccuracy Score:", accuracy\_score(y\_test, y\_pred))

# %% [code] {"execution":{"iopub.status.busy":"2024-12-18T15:52:05.640721Z","iopub.execute\_input":"2024-12-18T15:52:05.641349Z","iopub.status.idle":"2024-12-18T15:52:06.283553Z","shell.execute\_reply.started":"2024-12-18T15:52:05.641305Z","shell.execute\_reply":"2024-12-18T15:52:06.282363Z"}}

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay, classification\_report

# Plot Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=xgb\_model.classes\_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.show()

# Plot Classification Report

def plot\_classification\_report(cr):

cr = cr.split("\n")

classes = []

values = []

for line in cr[2:-5]:

parts = line.split()

classes.append(parts[0])

values.append(list(map(float, parts[1:4])))

fig, ax = plt.subplots()

sns.heatmap(values, annot=True, fmt=".2f", cmap="YlGnBu", xticklabels=["Precision", "Recall", "F1-Score"], yticklabels=classes, ax=ax)

plt.title("Classification Report")

plt.show()

cr = classification\_report(y\_test, y\_pred)

plot\_classification\_report(cr)