Configuration Manual

MSc Research Project

MSc Cloud Computing

Tejas Chavan

Student ID: X22206183

School of Computing

National College of Ireland

Supervisor: Aqeel Kazmi

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Tejas Chavan
Student ID:	…x22206183
Programme:	MSc in Cloud Computing
Year:	2023-2024
Module:	MSc Research Project
Lecturer:	Aqeel Kazmi
Submission Due Date:	03-01-2025
Project Title:	Enhancing Network Layer Security in Cloud Computing through Machine Learning Techniques
Word Count:	……………………………………… **Page Count**……………………………….…….………

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

Signature: Tejas Chavan

Date: 03-01-2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

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

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

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

Configuration Manual

Tejas Chavan
X22206183

1. Introduction

This Configuration Manual outlines the prerequisites and steps required to replicate the research study titled "Enhancing Network Layer Security in Cloud Computing through Machine Learning Techniques". The document covers system setup, data preprocessing, model implementation, and evaluation procedures. Screenshots and visualizations will accompany the steps where required and this config manual is organized as follows:

- System Requirements
- Data Collection
- Data Preprocessing
- Model Implementation
- Evaluation

---

2. System Requirements

2.1 Hardware Requirements

The hardware specifications necessary for the study are shown below:

- **CPU**: Minimum 8-core processor (e.g., Intel Core i7 or AMD Ryzen 7).
- **GPU**: NVIDIA GPU with CUDA support (e.g., NVIDIA RTX 3060 or higher).
- **RAM**: At least 16 GB.
- **Storage**: 200 GB of free storage space.

2.2 Software Requirements

- **Operating System**: Ubuntu 20.04 LTS or equivalent.
- **Programming Language**: Python 3.8 or higher.
- **Integrated Development Environment**: Jupyter Notebook (provided with Anaconda).

Figure 1: List of Required Libraries

```python
import os
import glob
import time
import joblib
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
from tqdm import tqdm
import matplotlib.pyplot as plt
from xgboost import XGBClassifier
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from imblearn.under_sampling import RandomUnderSampler
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix


plt.rcParams['figure.dpi'] = 300
warnings.filterwarnings('ignore')
```

2.3 Code Execution

1. Install Anaconda from https://www.anaconda.com/download
2. Launch Jupyter Notebook.
3. Navigate to the directory containing the code file.
4. Open the file and execute all cells sequentially.

---

3. Data Collection

The primary dataset used in this study is the CTU-13 dataset, obtained from the official CTU University of Prague repository. This dataset contains labeled network traffic data from multiple scenarios, making it ideal for training machine learning models for intrusion detection.

3.1 Dataset Details

- **Source**: Official Repository
- **File Format**: CSV
- **Size**: Approximately 20 million records.
- **Attributes**:
    - Traffic features (e.g., protocol, source and destination IPs, ports).
    - Labels indicating traffic types (e.g., benign, botnet, background).

Figure 2: Sample of CTU-13 Dataset

Table 2 – Characteristics of the botnet scenarios. (CF: ClickFraud, PS: Port Scan, FF: FastFlux, US: Compiled and controlled by us.)

Id	IRC	SPAM	CF	PS	DDoS	FF	P2P	US	HTTP	Note
1	√	√	√							
2	√	√	√							
3	√			√				√		
4	√				√			√		UDP and ICMP DDoS.
5		√		√					√	Scan web proxies.
6				√						Proprietary C&C. RDP.
7									√	Chinese hosts.
8				√						Proprietary C&C. Net-BIOS, STUN.
9	√	√	√	√						
10	√				√			√		UDP DDoS.
11	√				√			√		ICMP DDoS.
12							√			Synchronization.
13		√		√					√	Captcha. Web mail.

3.2 Loading the Dataset

- Load all CSV files using Python's glob and pandas libraries.
- Combine the files into a single DataFrame.
- Use the head() method to inspect the first few rows for verification.

Figure 3: Dataset Loading Code

```python
# Use glob to find all parquet files in the CTU-13 directory
all_files = glob.glob(os.path.join('CTU-13', '*.csv'))
# Initialize an empty list to hold DataFrames
df_list = []
# Iterate over each file and read it into a DataFrame
for file in all_files:
    df = pd.read_csv(file)
    df_list.append(df)
# Concatenate all DataFrames into a single DataFrame
CTU13 = pd.concat(df_list, ignore_index=True)
# Display the head of the CTU13 DataFrame
print('CTU-13 DataFrame Head:')
display(CTU13.head())
```

Python

CTU-13 DataFrame Head:

	StartTime	Dur	Proto	SrcAddr	Sport	Dir	DstAddr	Dport	State	sTos	dTos	TotPkts	TotB
0	2011/08/10 09:46:59.607825	1.026539	tcp	94.44.127.113	1577	->	147.32.84.59	6881	S_RA	0.0	0.0	4	
1	2011/08/10 09:47:00.634364	1.009595	tcp	94.44.127.113	1577	->	147.32.84.59	6881	S_RA	0.0	0.0	4	
2	2011/08/10 09:47:48.185538	3.056586	tcp	147.32.86.89	4768	->	77.75.73.33	80	SR_A	0.0	0.0	3	
3	2011/08/10 09:47:48.230897	3.111769	tcp	147.32.86.89	4788	->	77.75.73.33	80	SR_A	0.0	0.0	3	
4	2011/08/10 09:47:48.963351	3.083411	tcp	147.32.86.89	4850	->	77.75.73.33	80	SR_A	0.0	0.0	3	

4. Data Preprocessing

Preprocessing ensures data quality and compatibility with machine learning models. Key steps include:

4.1 Handling Missing Values

- Columns with significant missing values include Sport, Dport, State, sTos, and dTos.
- Drop rows with null values to maintain data integrity.

Figure 4: Missing Values Handling

```python
# Checking for and display columns with null values
print("Columns with null values:")
print(CTU13.isnull().sum())
```

```
Columns with null values:
StartTime          0
Dur                0
Proto              0
SrcAddr            0
Sport         203085
Dir                0
DstAddr            0
Dport         194062
State           1378
sTos          220525
dTos         1718011
TotPkts            0
TotBytes           0
SrcBytes           0
Label              0
dtype: int64
```

```python
# Check for and handle duplicate rows
print(f"Duplicate rows found: {CTU13.duplicated().sum()}")
```

```
Duplicate rows found: 0
```

```python
# Display summary statistics of the DataFrame
print("Summary statistics:")
display(CTU13.describe())
```

```
Summary statistics:
```

	Dur	sTos	dTos	TotPkts	TotBytes	SrcBytes
count	1.997670e+07	1.975618e+07	1.825869e+07	1.997670e+07	1.997670e+07	1.997670e+07
mean	2.879468e+02	8.158963e-02	4.074772e-04	4.139147e+01	3.232745e+04	6.435360e+03
std	8.318070e+02	3.910225e+00	3.245888e-02	5.545725e+03	3.983037e+06	1.667901e+06
min	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+00	6.000000e+01	0.000000e+00
25%	2.750000e-04	0.000000e+00	0.000000e+00	2.000000e+00	2.140000e+02	7.800000e+01
50%	7.270000e-04	0.000000e+00	0.000000e+00	2.000000e+00	2.640000e+02	8.300000e+01
75%	1.966033e+00	0.000000e+00	0.000000e+00	4.000000e+00	6.190000e+02	3.040000e+02
max	3.657061e+03	1.920000e+02	3.000000e+00	1.658064e+07	4.376239e+09	3.423408e+09

4.2 Feature Engineering

1. StartTime and Dir were removed as they are non-informative for prediction.
2. Original labels were merged into three categories: Background, Normal, and Botnet.
3. Columns like Proto, SrcAddr, and State were converted to numerical format using Label Encoding.

Figure 5: Feature Engineering Code

```python
# Remove rows with null values
CTU13 = CTU13.dropna()
print("\nNull values removed.")
# Display null counts again to confirm removal
print("\nColumns with null values and their counts after removal:")
print(CTU13.isnull().sum())

# Drop the 'StartTime' and 'Dir' columns
CTU13 = CTU13.drop(columns=['StartTime', 'Dir'])
# Display the head of the CTU13 DataFrame
print('\nCTU-13 DataFrame Head after Cleaning:')
display(CTU13.head())
# Display the remaining columns to confirm removal
print("\nRemaining columns after dropping 'StartTime' and 'Dir':")
display(CTU13.columns)

# Display the shape of the DataFrame
print("\nShape of the CTU13 DataFrame after Cleaning:", CTU13.shape)
```

```
Null values removed.

Columns with null values and their counts after removal:
StartTime    0
Dur          0
Proto        0
SrcAddr      0
Sport        0
Dir          0
DstAddr      0
Dport        0
State        0
sTos         0
dTos         0
TotPkts      0
TotBytes     0
SrcBytes     0
Label        0
dtype: int64
```

	Dur	Proto	SrcAddr	Sport	DstAddr	Dport	State	sTos	dTos	TotPkts	TotBytes	SrcBytes
0	1.026539	tcp	94.44.127.113	1577	147.32.84.59	6881	S_RA	0.0	0.0	4	276	156
1	1.009595	tcp	94.44.127.113	1577	147.32.84.59	6881	S_RA	0.0	0.0	4	276	156
2	3.056586	tcp	147.32.86.89	4768	77.75.73.33	80	SR_A	0.0	0.0	3	182	122
3	3.111769	tcp	147.32.86.89	4788	77.75.73.33	80	SR_A	0.0	0.0	3	182	122
4	3.083411	tcp	147.32.86.89	4850	77.75.73.33	80	SR_A	0.0	0.0	3	182	122

```
Remaining columns after dropping 'StartTime' and 'Dir':

Index(['Dur', 'Proto', 'SrcAddr', 'Sport', 'DstAddr', 'Dport', 'State', 'sTos',
       'dTos', 'TotPkts', 'TotBytes', 'SrcBytes', 'Label'],
      dtype='object')

Shape of the CTU13 DataFrame after Cleaning: (18048817, 13)
```

4.3 Data Balancing

- Random Under-Sampling (RUS) reduces the majority class size.
- SMOTE synthesizes new samples for the minority class.

Figure 6: Data Balancing Code

```python
# Separate features and target variable
X = CTU13.drop(columns=['Merged_Label'])  # Features
y = CTU13['Merged_Label']                # Target

# Encode object type (categorical) features
for column in X.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    X[column] = le.fit_transform(X[column])

# Define the target sample size
target_class_size = y.value_counts()['Botnet']

# Define the number of batches to avoid memory issues
num_batches = 20

# Split the data into smaller batches
X_batches = np.array_split(X, num_batches)
y_batches = np.array_split(y, num_batches)

# Create empty lists to store resampled data
X_resampled_list = []
y_resampled_list = []

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply under-sampling and SMOTE to each batch
for i in tqdm(range(num_batches), desc="Balancing Data in Batches"):
    # Extract the current batch
    X_batch = X_batches[i]
    y_batch = y_batches[i]

    # Dynamically calculate the under-sampling strategy for the current batch
    undersample_strategy = {'Background': min(y_batch.value_counts().get('Background', 0), target_class_si
                            'Normal': min(y_batch.value_counts().get('Normal', 0), target_class_size),
                            'Botnet': min(y_batch.value_counts().get('Botnet', 0), target_class_size)}

    # Apply under-sampling to the current batch
    undersample = RandomUnderSampler(sampling_strategy=undersample_strategy, random_state=42)
    X_under, y_under = undersample.fit_resample(X_batch, y_batch)
    # Apply SMOTE to balance the under-sampled data
    smote_strategy = {'Background': target_class_size,'Normal': target_class_size,'Botnet': target_class_s
    smote = SMOTE(sampling_strategy=smote_strategy, random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_under, y_under)

    # Append the resampled batch to the lists
    X_resampled_list.append(pd.DataFrame(X_resampled, columns=X.columns))
    y_resampled_list.append(pd.Series(y_resampled))

# Concatenate all the resampled batches to form the final balanced dataset
X_balanced = pd.concat(X_resampled_list, axis=0, ignore_index=True)
y_balanced = pd.concat(y_resampled_list, axis=0, ignore_index=True)

# Display the class distribution after balancing
print("Class distribution after balancing:")
print(y_balanced.value_counts())

# Combine features and labels into a single DataFrame for easy saving
CTU13_Balanced = pd.concat([pd.DataFrame(X_balanced, columns=X.columns), pd.Series(y_balanced, name='Merge

# Save the balanced dataset to a CSV file
CTU13_Balanced.to_csv('CTU13_Balanced.csv', index=False)
print("Balanced dataset saved to 'CTU13_Balanced.csv'.")
```

Category	Count
Background	4,237,660
Botnet	4,237,660
Normal	4,237,660

4.4 Normalization

- Use StandardScaler to normalize numerical features.

Figure 7: Normalization Code

```
# Separate features and target variable
X = CTU13_Balanced.drop(columns=['Merged_Label'])  # Features
y = CTU13_Balanced['Merged_Label']                 # Target

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Scale the feature data using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Print the shapes of the splits
print("Shape of Training Features:", X_train_scaled.shape)
print("Shape of Testing Features:", X_test_scaled.shape)
print("Shape of Training Labels:", y_train.shape)
print("Shape of Testing Labels:", y_test.shape)
```

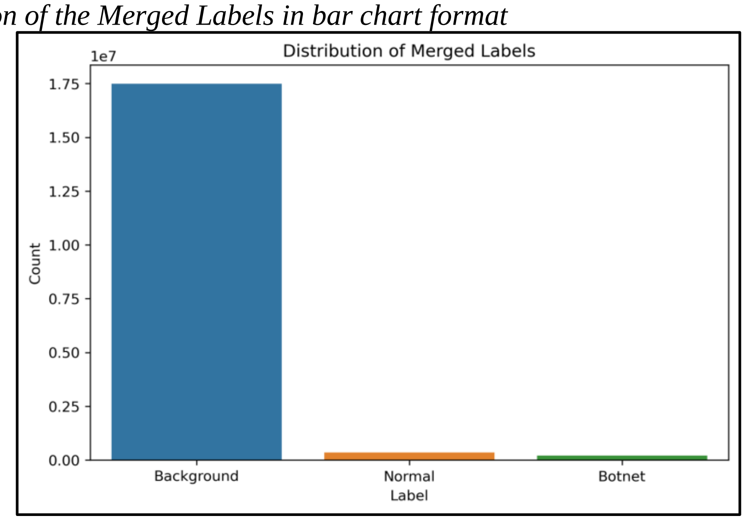
Dataset Component	**Shape**
Training Features	(10,170,384, 12)
Testing Features	(2,542,596, 12)
Training Labels	-10,170,384
Testing Labels	-2,542,596

---

5. EDA

The following plot reveals a clear imbalance in the dataset. The "Background" label has an overwhelmingly high count compared to "Normal" and "Botnet." This suggests that most of the traffic is classified as Background, with significantly fewer instances of Normal and Botnet traffic. Such an imbalance can impact model performance, as a model trained on this dataset may become biased toward predicting the Background label, highlighting the need for data-balancing methods.

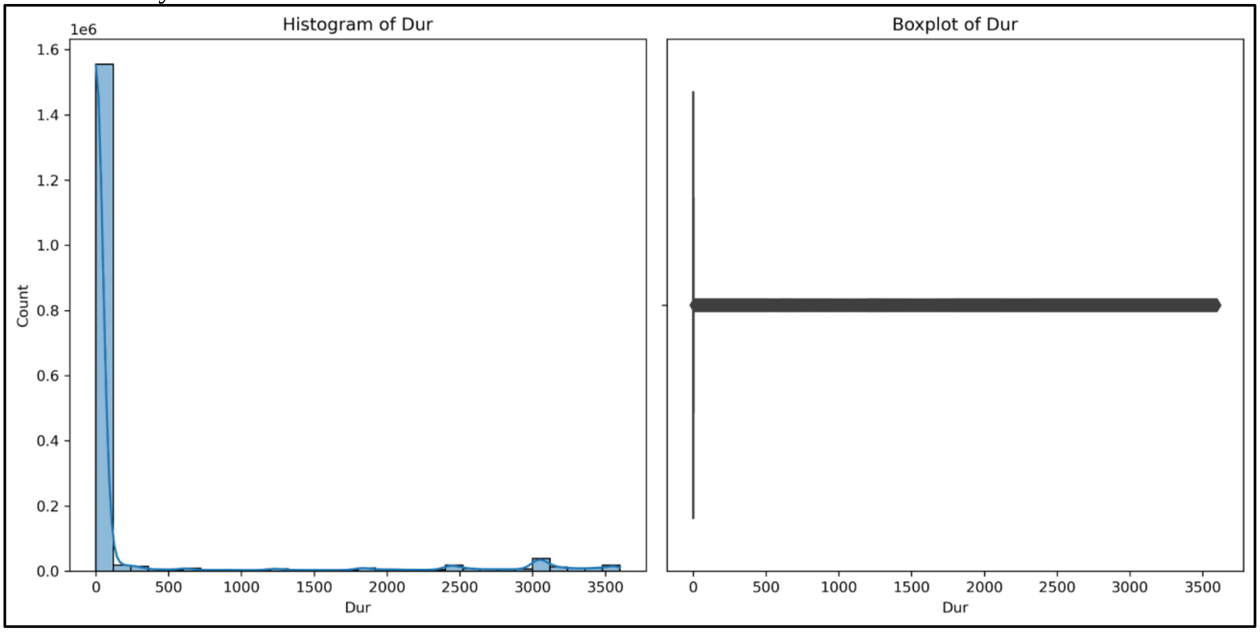
Figure 8: Distribution of the Merged Labels in bar chart format



The following histogram on the left side of each figure shows the distribution of values for each feature and the adjacent boxplot on the right provides a summary of the spread, quartiles, and potential outliers for each feature. In these plots, many data points are positioned as outliers, particularly beyond the upper quartile, indicated by dots. These following are histograms and boxplots of:

- Dur
- sTos

- dTos
- TotPkts
- TotBytes
- SrcBytes



In the protocol distribution plot, we observe that UDP and TCP are the most frequently used protocols, especially within the "Background" label, which is expected in typical network traffic. The "Normal" and "Botnet" traffic appears minimally within these protocols as follows.

Figure 9: Bar plot of Protocol Distribution by Label

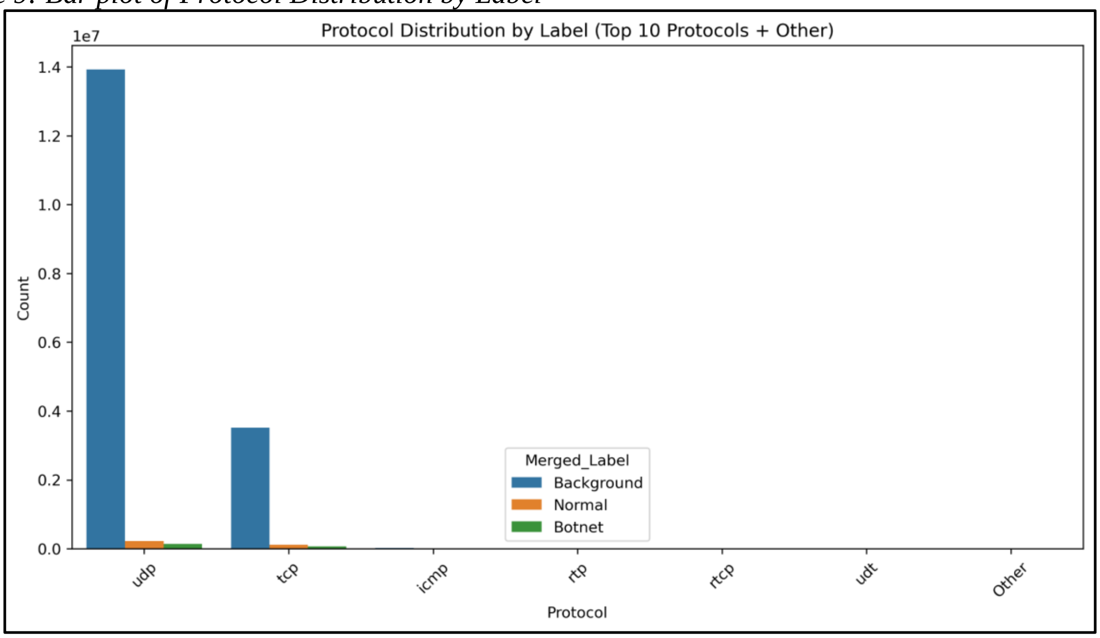
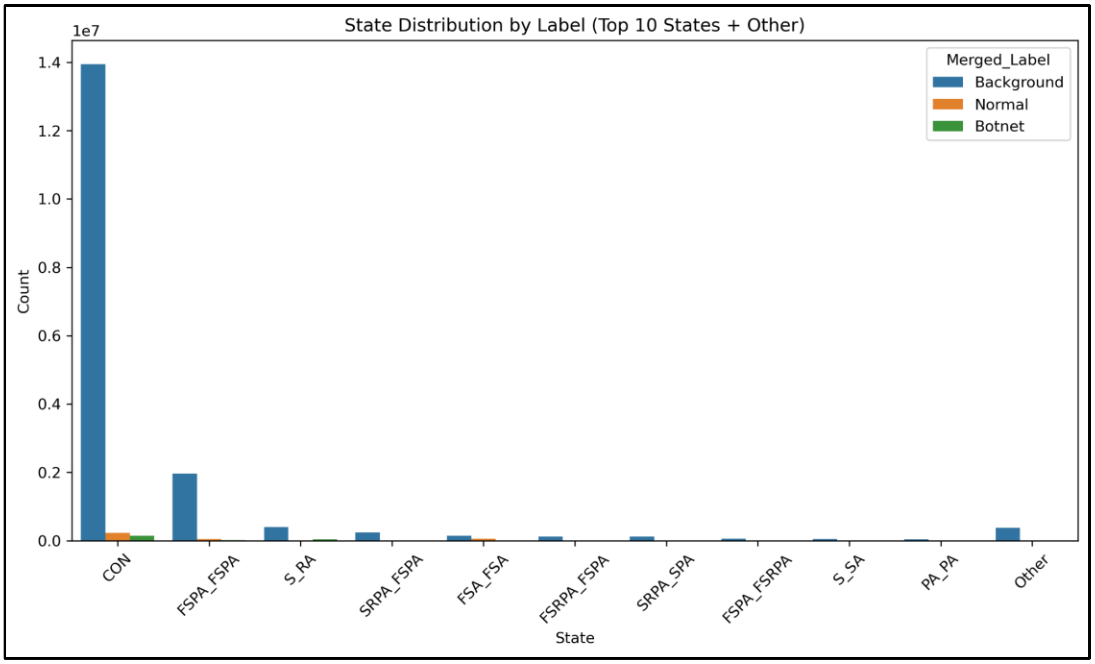


Figure 10: Bar plot of State Distribution by Label



6. Model Implementation

The following machine learning models were implemented:

6.1 Logistic Regression

- The purpose of the baseline classification model.
- Hyperparameters used are regularization strength C, penalty type.

Figure 11: Logistic Regression Code

```
# Initialize and train the Logistic Regression model
log_reg = LogisticRegression(max_iter=1000, random_state=42, verbose=2)

# Measure the training time
start_time = time.time()
log_reg.fit(X_train_scaled, y_train)
training_time = time.time() - start_time

print(f"Model training completed in {training_time:.2f} seconds.")

# Measure the prediction time
start_time = time.time()
y_pred = log_reg.predict(X_test_scaled)
detection_time = time.time() - start_time

print(f"Detection completed in {detection_time:.2f} seconds.")

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=log_reg.classes_)

# Calculate False Positive Rate (FPR)
# FPR = FP / (FP + TN) for each class
fpr = {}
for i, label in enumerate(log_reg.classes_):
    fp = cm[:, i].sum() - cm[i, i]  # False Positives for the class
    tn = cm.sum() - cm[i, :].sum() - cm[:, i].sum() + cm[i, i]  # True Negatives for the class
    fpr[label] = fp / (fp + tn) if (fp + tn) > 0 else 0

print("\nFalse Positive Rate (FPR) for each class:")
for label, rate in fpr.items():
    print(f"{label}: {rate:.4f}")

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=log_reg.classes_)
plt.figure(figsize=(8, 6))
disp.plot(cmap='Blues', values_format='d')
plt.title("Confusion Matrix")
plt.show()
```

Model Performance	
Metric	**Value**
Training Time	496.28 seconds
Detection Time	0.25 seconds

Classification Report				
Class	**Precision**	**Recall**	**F1-Score**	**Support**
Background	1	0.3783	0.4872	847,532
Botnet	1	0.872	0.8578	847,532
Normal	1	0.8556	0.7089	847,532
Accuracy			0.702	2,542,596
Macro Avg	0.7111	0.702	0.6846	2,542,596
Weighted Avg	0.7111	0.702	0.6846	2,542,596

6.2 Random Forest

- Ensemble learning model for robust classification.
- Hyperparameters:
  - Number of estimators: 100
  - Maximum depth: Auto

Figure 12: Random Forest Code

```python
# Initialize the Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42, verbose=2)

# Measure the training time
start_time = time.time()
rf.fit(X_train_scaled, y_train)
training_time = time.time() - start_time

print(f"Model training completed in {training_time:.2f} seconds.")

# Measure the prediction time
start_time = time.time()
y_pred = rf.predict(X_test_scaled)
detection_time = time.time() - start_time

print(f"Detection completed in {detection_time:.2f} seconds.")

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=rf.classes_)

# Calculate False Positive Rate (FPR)
# FPR = FP / (FP + TN) for each class
fpr = {}
for i, label in enumerate(rf.classes_):
    fp = cm[:, i].sum() - cm[i, i]  # False Positives for the class
    tn = cm.sum() - cm[i, :].sum() - cm[:, i].sum() + cm[i, i]  # True Negatives for the class
    fpr[label] = fp / (fp + tn) if (fp + tn) > 0 else 0

print("\nFalse Positive Rate (FPR) for each class:")
for label, rate in fpr.items():
    print(f"{label}: {rate:.4f}")

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf.classes_)
plt.figure(figsize=(8, 6))
disp.plot(cmap='Greens', values_format='d')
plt.title("Confusion Matrix")
plt.show()
```

Metric	Value

Training Time	4183.68 seconds (~69.3 minutes)
Detection Time	49.92 seconds
Accuracy	0.999
Precision (Macro Avg)	0.999
Recall (Macro Avg)	0.999
F1-Score (Macro Avg)	0.999
False Positive Rate	Background: 0.0006, Botnet: 0.0002, Normal: 0.0008

6.3 XGBoost

- Gradient boosting model optimized for speed and performance.
- Hyperparameters:
  - Learning rate: 0.1
  - Maximum depth: 3
  - Number of estimators: 100

Figure 13: XGBoost Code

```
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Initialize the XGBoost Classifier
xgb = XGBClassifier(n_estimators=100,learning_rate=0.1,max_depth=3,use_label_encoder=False,eval_metric='ml

# Measure the training time
start_time = time.time()
xgb.fit(X_train_scaled, y_train_encoded)
training_time = time.time() - start_time

print(f"Model training completed in {training_time:.2f} seconds.")

# Measure the prediction time
start_time = time.time()
y_pred_encoded = xgb.predict(X_test_scaled)
detection_time = time.time() - start_time

print(f"Detection completed in {detection_time:.2f} seconds.")

# Decode the predicted labels back to the original class names
y_pred = label_encoder.inverse_transform(y_pred_encoded)

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=label_encoder.classes_)

# Calculate False Positive Rate (FPR)
# FPR = FP / (FP + TN) for each class
fpr = {}
for i, label in enumerate(label_encoder.classes_):
    fp = cm[:, i].sum() - cm[i, i]  # False Positives for the class
    tn = cm.sum() - cm[i, :].sum() - cm[:, i].sum() + cm[i, i]  # True Negatives for the class
    fpr[label] = fp / (fp + tn) if (fp + tn) > 0 else 0

print("\nFalse Positive Rate (FPR) for each class:")
for label, rate in fpr.items():
    print(f"{label}: {rate:.4f}")

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=label_encoder.classes_)
plt.figure(figsize=(8, 6))
disp.plot(cmap='Greys', values_format='d')
```

Metric	Value

Training Time	150.68 seconds
Detection Time	2.07 seconds
Accuracy	0.9899
Precision (Macro Avg)	0.9899
Recall (Macro Avg)	0.9899
F1-Score (Macro Avg)	0.9899
False Positive Rate	Background: 0.0055, Botnet: 0.0031, Normal: 0.0065

6.4 Gradient Boosting Machine (GBM)

- Sequential ensemble model for high predictive accuracy.
- Hyperparameters:
    - Learning rate: 0.1
    - Number of estimators: 100
    - Maximum depth: 3

Figure 14: Gradient Boosting Machine Code

```python
# Initialize the Gradient Boosting Classifier
gbm = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3, random_state=42, verbos

# Measure the training time
start_time = time.time()
gbm.fit(X_train_scaled, y_train)
training_time = time.time() - start_time

print(f"Model training completed in {training_time:.2f} seconds.")

# Measure the prediction time
start_time = time.time()
y_pred = gbm.predict(X_test_scaled)
detection_time = time.time() - start_time

print(f"Detection completed in {detection_time:.2f} seconds.")

# Print the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, digits=4))

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=gbm.classes_)

# Calculate False Positive Rate (FPR)
# FPR = FP / (FP + TN) for each class
fpr = {}
for i, label in enumerate(gbm.classes_):
    fp = cm[:, i].sum() - cm[i, i]  # False Positives for the class
    tn = cm.sum() - cm[i, :].sum() - cm[:, i].sum() + cm[i, i]  # True Negatives for the class
    fpr[label] = fp / (fp + tn) if (fp + tn) > 0 else 0

print("\nFalse Positive Rate (FPR) for each class:")
for label, rate in fpr.items():
    print(f"{label}: {rate:.4f}")

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=gbm.classes_)
plt.figure(figsize=(8, 6))
disp.plot(cmap='Reds', values_format='d')
plt.title("Confusion Matrix")
plt.show()
```

Metric	Value
Training Time	8759.19 seconds (~146 minutes)

Detection Time	12.47 seconds
Accuracy	0.9941
Precision (Macro Avg)	0.9941
Recall (Macro Avg)	0.9941
F1-Score (Macro Avg)	0.9941
False Positive Rate	Background: 0.0033, Botnet: 0.0019, Normal: 0.0037

7. Evaluation

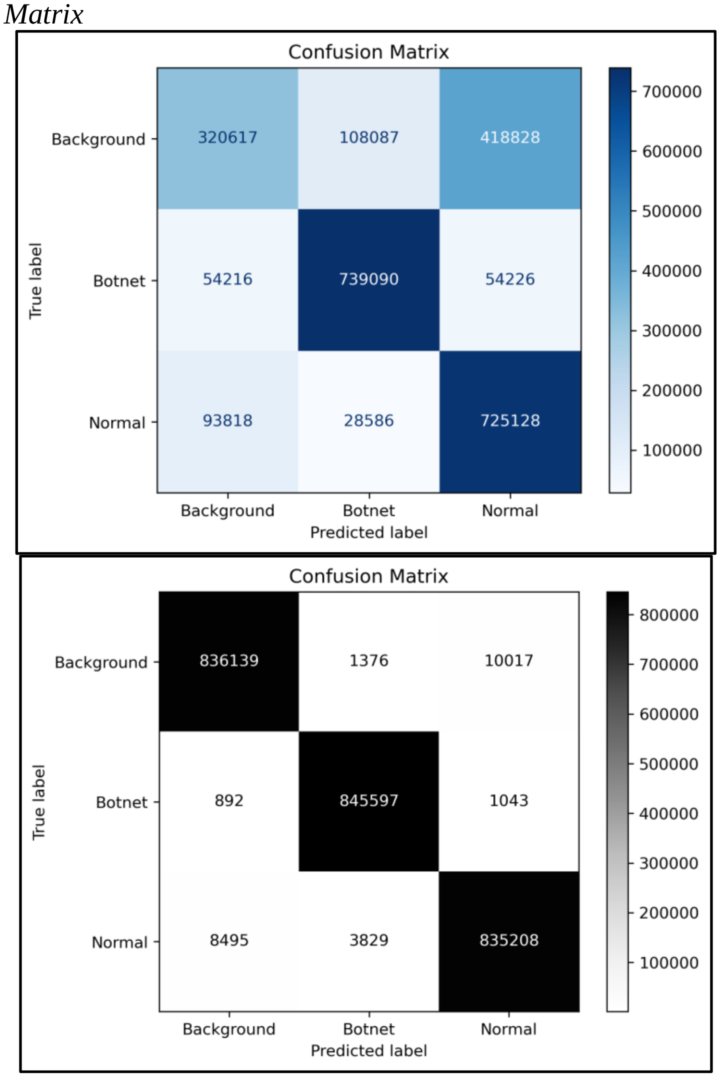
Model performance was evaluated using the following metrics:

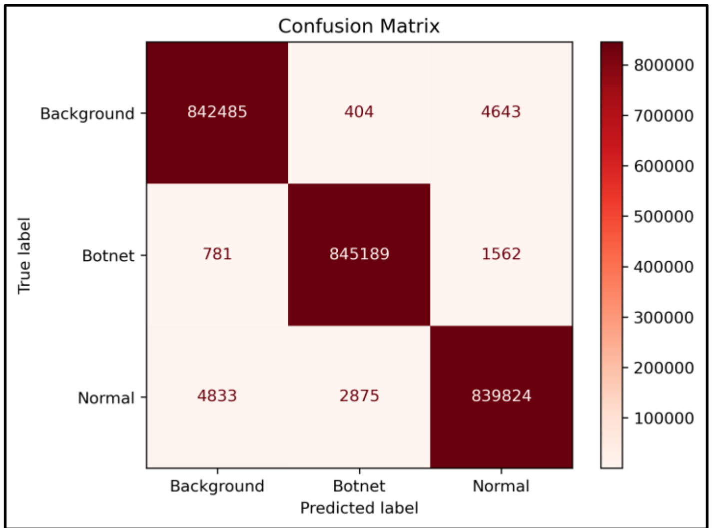
- Accuracy: Overall correctness.
- Precision: True positive rate among predicted positives.
- Recall: Sensitivity to actual positives.
- F1-Score: Harmonic means of precision and recall.

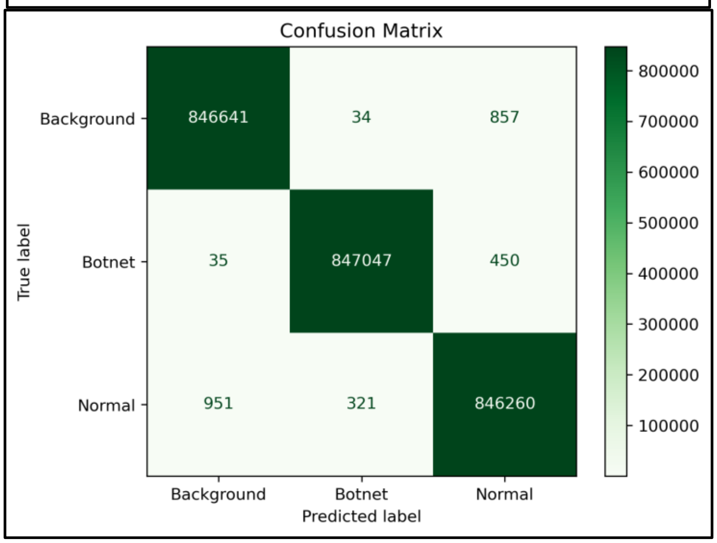
7.1 Confusion Matrices

Visualizations of prediction outcomes for each class provide insights into misclassifications.

Figure 15: Confusion Matrix





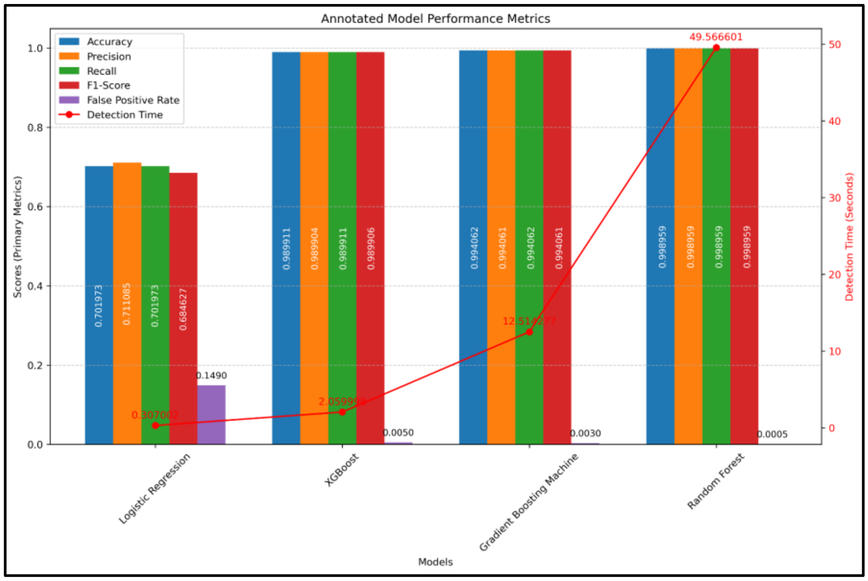


7.2 Performance Metrics

The table below summarizes key performance metrics for each model:

Model	Accuracy	Precision	Recall	F1-Score	FPR	Detection Time
Logistic Regression	70.20%	71.11%	70.20%	68.46%	12.33%	0.25
XGBoost	98.99%	98.99%	98.99%	98.99%	1.50%	2.07
Gradient Boosting	99.41%	99.41%	99.41%	99.41%	0.49%	12.47
Random Forest	99.90%	99.90%	99.90%	99.90%	0.18%	49.92

Figure 16: Performance Metrics



8. Conclusion

This manual serves as a guide to replicate the research study, ensuring reproducibility and clarity in methodology. For more details, refer to the full research paper.

References

- CTU-13 Dataset: https://www.stratosphereips.org/datasets-ctu13
- Python Libraries Documentation:
  - Pandas
  - Scikit-learn
  - XGBoost