Configuration Manual

MSc Practicum 2
Master of Science in Cybersecurity

Preetham Charan Sridhar
Student ID: x23183683

School of Computing
National College of Ireland

Supervisor: Vikas Sahni

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	PREETHAM CHARAN SRIDHAR
Student ID:	x23183683
Programme:	MSC Cyber Security **Year:** 2024
Module:	MSc Practicum 2
Lecturer:	VIKAS SAHNI
Submission Due Date:	12-12-2024
Project Title:	Securing 5G IoT Networks: A Machine Learning Framework for Zero-Trust Intrusion Detection System
Word Count:	2126 **Page Count:** 16

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: PREETHAM CHARAN SRIDHAR

Date: 12-12-2024

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

Preetham Charan Sridhar
Student ID: x23183683

1. Introduction

This configuration manual walks through the implementations of machine learning models for intrusion detection in 5G IoT networks. It details the packages, libraries, and steps needed for data preprocessing, model training, and evaluation. In fact, this manual supports replication experiments-including those with hybrid models and Zero Trust integration-to improve IoT network security.

2. Hardware requirements/environment used.

Cloud Hardware

- **RAM** – up to 51GB
- **Disk Storage** - 225.8 GB available for temporary file storage during runtime
- **Compute Units -** 100 units for operations
- **GPU -** NVIDIA Tesla P100 or T4 (CUDA-enabled) for accelerated training
- **Backend -** Python 3 Google Compute Engine
- **Internet Connection** - High-speed internet for seamless access to the Colab Pro environment and dataset uploads.

3. Software Requirements

Software tools and environment used,

- **Cloud Platform** - Google Colab Pro.
- **Operating System** - Not required locally, runs entirely on Colab Pro backend.
- **Programming Language** - Python 3.8+ (pre-installed in Colab Pro).
- **Development Environment -** Jupyter Notebook interface (via Colab Pro).
- **Data Integration -** Google Drive integration for seamless dataset management.

4. Datasets Management

The datasets (CICIOT2023 and Bot-IoT) are utilized in the research, and these datasets were uploaded and accessed via Google Drive within Colab Pro.

5. Libraries Imported

The library list is detailed and categorized into Data Loading, Manipulation, and Preprocessing and Machine Learning Algorithms.

Data Loading, Manipulation, and Preprocessing

Libraries and Versions

- Pandas (2.2.2)
- NumPy (1.26.4)
- Scikit-learn (1.5.2)
- XGBoost (2.1.3)
- Matplotlib (3.8.0)
- Seaborn (0.13.2)
- Imbalanced-learn (0.12.4)

6. Data Preparation and Processing stage

CICIOT2023 - Data Preparation and Processing Stage.

Step 1. Mounting and Loading - The data was accessed directly from Google Drive. Renaming columns for consistency: all in lower case and separated by underscores.
Step 2. Data Cleaning - some irrelevant columns, like the protocol type, have been removed. Missing values have also been checked and none found.

```
from google.colab import drive
import pandas as pd

# Mount Google Drive to access the dataset
drive.mount('/content/drive')

# Define the path to your CSV file
CSV_PATH = '/content/drive/MyDrive/Malware1/CICIOT2023.csv'

# Load the data
df = pd.read_csv(CSV_PATH)

# Rename columns to lowercase and replace spaces with underscores
df.columns = df.columns.str.replace(" ", "_").str.lower()
print("Columns renamed for consistency:")
print(df.columns)

# Display basic information about the dataset
df.info()
```

Step 1. Mounting and Loading

```
# Fix known typos in column names if needed
df = df.rename(columns={"magnitue": "magnitude"})

# Check for missing values in each column
missing_values = df.isnull().sum()
print("Missing values in each column:\n", missing_values[missing_values > 0])

# Drop unnecessary columns (e.g., protocol_type if it's not used)
df = df.drop(columns=['protocol_type'], errors='ignore')

# Display updated info to confirm cleaning
df.info()
```

Step 2. Data Cleaning

Step 3. Feature Scaling - This step scales numerical features into a consistent range using Standardscaler.
Step 4. Class Balancing - The classes were then balanced by SMOTE, after which the class distributions for all attack categories were equal.

```
from sklearn.preprocessing import StandardScaler

# Separate features and labels
features = df.drop(columns=['label'])
labels = df['label']

# Apply StandardScaler to the features
scaler = StandardScaler()
scaled_features = pd.DataFrame(scaler.fit_transform(features), columns=features.columns)

# Combine scaled features with labels
scaled_df = pd.concat([scaled_features, labels.reset_index(drop=True)], axis=1)
print("Data scaling complete.")

Data scaling complete.
```

Step 3. Data Scaling

```
from imblearn.over_sampling import SMOTE

# Separate features and labels again after scaling
X = scaled_df.drop(columns=['label'])
y = scaled_df['label']

# Apply SMOTE to balance classes
smote = SMOTE(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Create a new DataFrame with the resampled data
balanced_df = pd.DataFrame(X_resampled, columns=X.columns)
balanced_df['label'] = y_resampled
print("Class balancing with SMOTE complete.")
print(balanced_df['label'].value_counts())
```

Step 4. Class Balancing

Step 5. Feature Correlation - Correlation heatmap has been generated to identify relationships between the features and removed redundant ones if needed.

Step 6. Label Encoding and Splitting - Labels were mapped to broader categories, encoded using LabelEncoder.

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Drop the non-numeric column 'label' from the dataframe for correlation
numeric_df = balanced_df.drop(columns=['label'])

# Generate a heatmap for feature correlations
plt.figure(figsize=(12, 10))
sns.heatmap(numeric_df.corr(), annot=False, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()
```

Step 5. Feature Correlation

```python
from sklearn.preprocessing import LabelEncoder

# Map labels to broader categories
remap_labels = {
    'DDoS-RSTFINFlood': 'DDoS', 'DDoS-PSHACK_Flood': 'DDoS', 'DDoS-SYN_Flood': 'DDoS',
    'DDoS-UDP_Flood': 'DDoS', 'DDoS-TCP_Flood': 'DDoS', 'DDoS-ICMP_Flood': 'DDoS',
    'DDoS-SynonymousIP_Flood': 'DDoS', 'DDoS-ACK_Fragmentation': 'DDoS',
    'DDoS-UDP_Fragmentation': 'DDoS', 'DDoS-ICMP_Fragmentation': 'DDoS',
    'DDoS-SlowLoris': 'DDoS', 'DDoS-HTTP_Flood': 'DDoS',
    'DoS-UDP_Flood': 'DoS', 'DoS-SYN_Flood': 'DoS', 'DoS-TCP_Flood': 'DoS',
    'DoS-HTTP_Flood': 'DoS', 'Mirai-greeth_flood': 'Mirai', 'Mirai-greip_flood': 'Mirai',
    'Mirai-udpplain': 'Mirai', 'Recon-PingSweep': 'Recon', 'Recon-OSScan': 'Recon',
    'Recon-PortScan': 'Recon', 'VulnerabilityScan': 'Recon', 'Recon-HostDiscovery': 'Recon',
    'DNS_Spoofing': 'Spoofing', 'MITM-ArpSpoofing': 'Spoofing', 'BenignTraffic': 'Benign',
    'BrowserHijacking': 'Web', 'Backdoor_Malware': 'Web', 'XSS': 'Web',
    'Uploading_Attack': 'Web', 'SqlInjection': 'Web', 'CommandInjection': 'Web',
    'DictionaryBruteForce': 'BruteForce'
}

# Apply mapping and encode labels
balanced_df['label'] = balanced_df['label'].map(remap_labels)
label_encoder = LabelEncoder()
balanced_df['label'] = label_encoder.fit_transform(balanced_df['label'])
print("Encoded label distribution:\n", balanced_df['label'].value_counts())
```

Step 6. Labe Encoding

Step 7. Dataset Splitting - The dataset was split into training (80%) and testing (20%) sets.

```python
from sklearn.model_selection import train_test_split

# Separate features and labels again if needed
X = balanced_df.drop(columns=['label'])
y = balanced_df['label']

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

print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")
```

Step 7. Dataset Splitting

BoT_IoT - Data Preparation and Processing Stage

Step 1. Data Loading - The dataset was loaded from Google Drive using pandas, and its structure was analysed to understand column names and data types.

Step 2. Data Cleaning – The data cleaning has been done for removing irrelevant columns (saddr, daddr, proto, etc.) and ARP packets and along with the confirmation of the cleaned dataset.

```python
# Import necessary libraries
import pandas as pd
import numpy as np

# Define paths to your data files in Google Drive
train_path = '/content/drive/My Drive/Bot_IoT/UNSW_2018_IoT_Botnet_Final_10_best_Training.csv'
test_path = '/content/drive/My Drive/Bot_IoT/UNSW_2018_IoT_Botnet_Final_10_best_Testing.csv'

# Load the datasets
train_data = pd.read_csv(train_path)
test_data = pd.read_csv(test_path)

# Check the shape of the loaded data
print("Train Data Shape:", train_data.shape)
print("Test Data Shape:", test_data.shape)
train_data.head(), test_data.head()
```

Step 1. Data Loading.

```python
#ARP packets were excluded because they are used to convert an IP address to a MAC address and irrelevant to the proposed attacks in the dataset
train_data = train_data[train_data.proto!="arp"]
test_data = test_data[test_data.proto!="arp"]

#dropping unused columns
train_data.drop(['pkSeqID', 'proto', 'saddr', 'sport', 'daddr', 'dport',"attack","category"],axis=1,inplace=True)
test_data.drop(['pkSeqID', 'proto', 'saddr', 'sport', 'daddr', 'dport',"attack","category"],axis=1,inplace=True)
full_data = pd.concat([train_data,test_data],axis=0)

# Select samples of "Data_Exfiltration" from train_data
data_exf = train_data.loc[train_data.subcategory == "Data_Exfiltration"].sample(n=2, random_state=42)

# Append the sampled data to test_data using pd.concat
test_data = pd.concat([test_data, data_exf])

# Drop the selected samples from train_data
train_data = train_data.drop(data_exf.index)

# Place the class in the begining of the dataframe
class_train = train_data['subcategory']
train_data.drop(['subcategory'], axis=1, inplace=True)
train_data.insert(0, 'subcategory', class_train)

class_test = test_data['subcategory']
test_data.drop(['subcategory'], axis=1, inplace=True)
test_data.insert(0, 'subcategory', class_test)

class_full = full_data['subcategory']
full_data.drop(['subcategory'], axis=1, inplace=True)
full_data.insert(0, 'subcategory', class_full)
```

Step 2 – Data Cleaning

Step 3. Class Distribution Analysis – The below bar chart shows the class distribution of attack subcategories in the unbalanced dataset (e.g., UDP, TCP, Keylogging, etc.)

```
d = full_data.subcategory.value_counts()
fig = px.bar(d, x=d.index, y=d.values,title = 'Class distribution between attack subcategories on full data',labels = {'index':'Attack','y':'Volume'},color=d.values)
fig.update_layout(title_x=0.5,width=1000, height=400)
fig.show()
```

Step 3. Class Distribution Analysis

Step 4. Balancing the Dataset – The below code is done for the undersampling the dominant class (DoS&DDoS) and balancing it with Service_Scan.

```
shuffled_df = full_data.sample(frac=1,random_state=4)

nondos_df = shuffled_df.loc[shuffled_df['subcategory'] != "DoS&DDoS"]

dos_df = shuffled_df.loc[shuffled_df['subcategory'] == "DoS&DDoS"].sample(n=73122,random_state=42)

normalized_full_df = pd.concat([nondos_df, dos_df])

#TRAIN after undersampling
d = normalized_full_df.subcategory.value_counts()
fig = px.bar(d, x=d.index, y=d.values,title = 'Class Label Distribution in Bot-IoT (undersampled)',labels = {'index':'Attack','y':'Volume'},color=d.values,text_auto='.2s')
fig.update_layout(title_x=0.5,width=1000, height=400)
fig.show()
```

Step 4. Balancing the Dataset

Step 5: Label Encoding and Data Splitting – The code preprocesses the dataset by dropping unnecessary columns and encoding categorical features using one-hot encoding. Using LabelEncoder, the target variables are converted to numerical values and features are normalized using StandardScaler. At last, the dataset was split into training and testing sets (80:20), while maintaining class proportions, and the class distribution is verified with Counter.

```
X = normalized_full_df.drop(["subcategory"], axis = 1)
##dropping features
# X.drop(['state_number'],axis=1,inplace=True)

Y = normalized_full_df['subcategory']

X = pd.get_dummies(X, prefix_sep='_')
le = LabelEncoder()
Y2 = le.fit_transform(Y)
X2 = StandardScaler().fit_transform(X)

X_Train, X_Test, Y_Train, Y_Test = train_test_split(X2, Y2, test_size = 0.2, random_state = 10)
```

```
print(X_Train.shape,X_Test.shape)
```

```
(131704, 10) (32927, 10)
```

```
print(Y_Train.shape,Y_Test.shape)
```

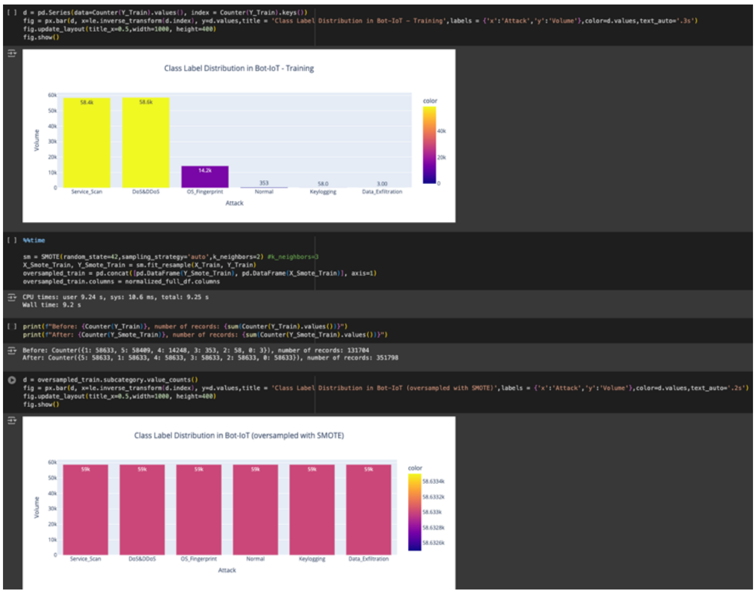
```
(131704,) (32927,)
```

```
print(Counter(Y_Test))
print(Counter(Y_Train))
```

```
Counter({5: 14713, 1: 14489, 4: 3634, 3: 77, 2: 12, 0: 2})
Counter({1: 58633, 5: 58409, 4: 14248, 3: 353, 2: 58, 0: 3})
```

Step 5: Label Encoding and Data Splitting

Step 6. Class Balancing with SMOTE – The Training dataset's class distribution was visualized to highlight imbalances. The SMOTE was then applied to balance the classes, and a bar chart confirmed the uniform distribution of all classes post-oversampling.



Step 6. Class Balancing with SMOTE

Step 7. Feature Correlation Analysis – Heatmap are generated to visualize feature correlations in both imbalanced and normalized datasets, helping us to identify relationships between features and reducing redundancy.

```
# Select only numeric columns for correlation calculation
numeric_cols = normalized_full_df.select_dtypes(include=['float64', 'int64'])

# Sample figsize in inches
fig, ax = plt.subplots(figsize=(12, 6))
fig, ax1 = plt.subplots(figsize=(12, 6))

# Imbalanced DataFrame Correlation
corr = numeric_cols.corr()
sns.heatmap(corr, cmap="YlGnBu", annot_kws={"size": 30}, ax=ax)
ax.set_title("Imbalanced Correlation Matrix", fontsize=14)

# Correlation Matrix after Normalization
corr2 = numeric_cols.corr()
sns.heatmap(corr2, cmap="YlGnBu", annot_kws={"size": 30}, ax=ax1)
ax1.set_title("Normalized Correlation Matrix", fontsize=14)

plt.show()
```

Step 7. Feature Correlation Analysis

7. Model Training Process

7.1 Base Classifiers Configuration.

CICIOT2023 - The base classifiers were trained to use the preprocessed dataset, with specific hyperparameter tuning applied to optimize performance. Logistic Regression used maximum iterations of 100 and random_state as 42, and where the KNN had n_neighbors of 5 to balance the accuracy and runtime. The Random Forest was using 50 estimators, and a maximum depth of 10 and parallel processing (n_jobs=-1). Naive Bayes (Gaussian) was operating on the default setting and the Decision Tree was configured at a depth of 10 to avoid overfitting. All the base classifiers were using the StratifiedKFold (3 splits) for cross-validation to ensure the model's effectiveness across the multiple folds.

```
# Logistic Regression
print("\nTraining Logistic Regression...")
start_time = time.time()
logistic_model = LogisticRegression(max_iter=100, random_state=42)
logistic_model.fit(X_train, y_train)
y_pred = logistic_model.predict(X_test)
end_time = time.time()
model_runtimes["Logistic Regression"] = end_time - start_time
print(f"Logistic Regression Runtime: {model_runtimes['Logistic Regression']:.4f} seconds")
print(f"Logistic Regression Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix for Logistic Regression")
plt.show()

# Cross-validation for Logistic Regression
cross_validate_model(logistic_model, X, y, "Logistic Regression")
```

Logistic Regression

```
# K-Nearest Neighbors (optional, remove if slow)
print("\nTraining K-Nearest Neighbors...")
start_time = time.time()
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)
y_pred = knn_model.predict(X_test)
end_time = time.time()
model_runtimes["K-Nearest Neighbors"] = end_time - start_time
print(f"K-Nearest Neighbors Runtime: {model_runtimes['K-Nearest Neighbors']:.4f} seconds")
print(f"K-Nearest Neighbors Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix for K-Nearest Neighbors")
plt.show()

# Cross-validation for K-Nearest Neighbors
cross_validate_model(knn_model, X, y, "K-Nearest Neighbors")
```

K-Nearest Neighbors

```
# Random Forest with reduced n_estimators and parallel processing
print("\nTraining Random Forest...")
start_time = time.time()
rf_model = RandomForestClassifier(n_estimators=50, max_depth=10, random_state=42, n_jobs=-1)
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
end_time = time.time()
model_runtimes["Random Forest"] = end_time - start_time
print(f"Random Forest Runtime: {model_runtimes['Random Forest']:.4f} seconds")
print(f"Random Forest Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix for Random Forest")
plt.show()

# Cross-validation for Random Forest (use X_sample, y_sample if dataset is very large)
cross_validate_model(rf_model, X, y, "Random Forest")
```

Random Forest

```
# Naive Bayes
print("\nTraining Naive Bayes...")
start_time = time.time()
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)
y_pred = nb_model.predict(X_test)
end_time = time.time()
model_runtimes["Naive Bayes"] = end_time - start_time
print(f"Naive Bayes Runtime: {model_runtimes['Naive Bayes']:.4f} seconds")
print(f"Naive Bayes Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix for Naive Bayes")
plt.show()

# Cross-validation for Naive Bayes
cross_validate_model(nb_model, X, y, "Naive Bayes")
```

Naive Bayes

```
# Decision Tree with limited depth to avoid overfitting
print("\nTraining Decision Tree...")
start_time = time.time()
dt_model = DecisionTreeClassifier(max_depth=10, random_state=42)
dt_model.fit(X_train, y_train)
y_pred = dt_model.predict(X_test)
end_time = time.time()
model_runtimes["Decision Tree"] = end_time - start_time
print(f"Decision Tree Runtime: {model_runtimes['Decision Tree']:.4f} seconds")
print(f"Decision Tree Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title("Confusion Matrix for Decision Tree")
plt.show()

# Cross-validation for Decision Tree (use X_sample, y_sample if dataset is very large)
cross_validate_model(dt_model, X, y, "Decision Tree")
```

Decision Tree

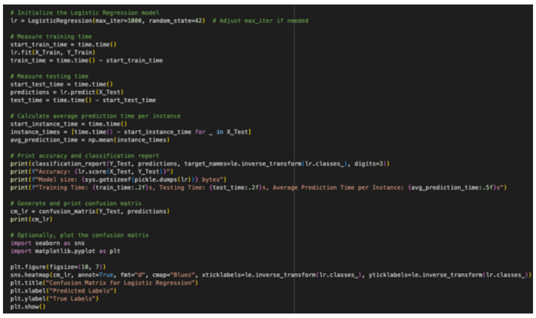
BoT_IoT - Random forest was configured with the n_estimators as 10 and fixed random_state=42 for maintaining the consistency, measuring and training and testing times. Logistic Regression used maximum iteration of 1000 to ensure convergence, while SVM applied a linear kernel for computational efficiency. KNN used n_neighbors as 5 to analyze local data relationships, and Gaussian Naive Bayes used its simplicity for probabilistic classification.The decision tree were optimized with maximum depth of 10 to prevent from the overfitting. The base classifiers were evaluated using the 5-fold cross-validation for effective accuracy comparisons.

```
# Initialize the Decision Tree model
dtc = DecisionTreeClassifier(random_state=42, max_depth=10)  # Adjust max_depth as needed

# Measure training time
start_train_time = time.time()
dtc.fit(X_Train, Y_Train)
train_time = time.time() - start_train_time

# Measure prediction time
start_test_time = time.time()
prediction_tree = dtc.predict(X_Test)
test_time = time.time() - start_test_time

# Print accuracy and classification report
print(classification_report(Y_Test, prediction_tree, target_names=le.inverse_transform(dtc.classes_), digits=3))
print(f"Accuracy: {dtc.score(X_Test, Y_Test):.3f}")
print(f"Model size: {sys.getsizeof(pickle.dumps(dtc))} bytes")
print(f"Training Time: {train_time:.2f}s, Prediction Time: {test_time:.2f}s")

# Generate and print confusion matrix
cm_tree = confusion_matrix(Y_Test, prediction_tree)
print(cm_tree)

# Optionally, plot the confusion matrix
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 7))
sns.heatmap(cm_tree, annot=True, fmt="d", cmap="Blues", xticklabels=le.inverse_transform(dtc.classes_), yticklabels=le.inverse_transform(dtc.classes_))
plt.title("Confusion Matrix for Decision Tree")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

Decision Tree

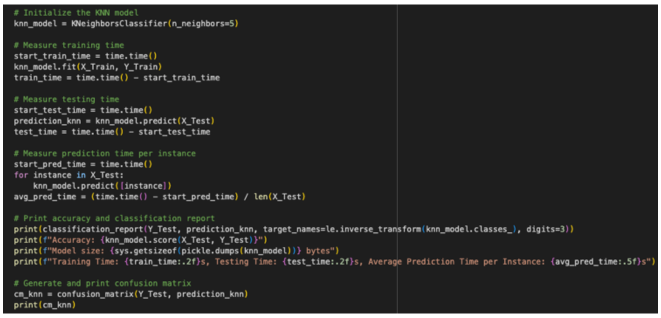
```
# Train Random Forest Classifier
start_train = time.time()
rfc = RandomForestClassifier(n_estimators=10, random_state=42).fit(X_Train, Y_Train)  # Ensure consistent random state
train_time = time.time() - start_train

# Measure prediction time
start_test = time.time()
prediction_forest = rfc.predict(X_Test)  # Store predictions in prediction_forest
test_time = time.time() - start_test

# Print classification report, accuracy, model size, training and prediction time
print(classification_report(Y_Test, prediction_forest, target_names=le.inverse_transform(rfc.classes_), digits=3))
print(f"Accuracy: {rfc.score(X_Test, Y_Test):.3f}")
print(f"Model size: {sys.getsizeof(pickle.dumps(rfc))} bytes")
print(f"Training Time: {train_time:.2f}s, Prediction Time: {test_time:.2f}s")

# Generate and print confusion matrix
cm_forest = confusion_matrix(Y_Test, prediction_forest)
print(cm_forest)
```

Random Forest



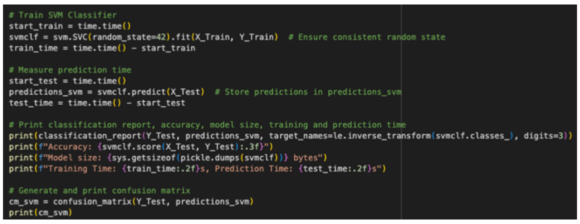
```python
# Initialize the Logistic Regression model
lr = LogisticRegression(max_iter=1000, random_state=42)  # Adjust max_iter if needed

# Measure training time
start_train_time = time.time()
lr.fit(X_Train, Y_Train)
train_time = time.time() - start_train_time

# Measure testing time
start_test_time = time.time()
predictions = lr.predict(X_Test)
test_time = time.time() - start_test_time

# Calculate average prediction time per instance
start_instance_time = time.time()
instance_times = [time.time() - start_instance_time for _ in X_Test]
avg_prediction_time = np.mean(instance_times)

# Print accuracy and classification report
print(classification_report(Y_Test, predictions, target_names=le.inverse_transform(lr.classes_), digits=3))
print(f"Accuracy: {lr.score(X_Test, Y_Test)}")
print(f"Model size: {sys.getsizeof(pickle.dumps(lr))} bytes")
print(f"Training Time: {train_time:.2f}s, Testing Time: {test_time:.2f}s, Average Prediction Time per Instance: {avg_prediction_time:.5f}s")

# Generate and print confusion matrix
cm_lr = confusion_matrix(Y_Test, predictions)
print(cm_lr)

# Optionally, plot the confusion matrix
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 7))
sns.heatmap(cm_lr, annot=True, fmt="d", cmap="Blues", xticklabels=le.inverse_transform(lr.classes_), yticklabels=le.inverse_transform(lr.classes_))
plt.title("Confusion Matrix for Logistic Regression")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

Logistic Regression



```python
# Initialize the KNN model
knn_model = KNeighborsClassifier(n_neighbors=5)

# Measure training time
start_train_time = time.time()
knn_model.fit(X_Train, Y_Train)
train_time = time.time() - start_train_time

# Measure testing time
start_test_time = time.time()
prediction_knn = knn_model.predict(X_Test)
test_time = time.time() - start_test_time

# Measure prediction time per instance
start_pred_time = time.time()
for instance in X_Test:
    knn_model.predict([instance])
avg_pred_time = (time.time() - start_pred_time) / len(X_Test)

# Print accuracy and classification report
print(classification_report(Y_Test, prediction_knn, target_names=le.inverse_transform(knn_model.classes_), digits=3))
print(f"Accuracy: {knn_model.score(X_Test, Y_Test)}")
print(f"Model size: {sys.getsizeof(pickle.dumps(knn_model))} bytes")
print(f"Training Time: {train_time:.2f}s, Testing Time: {test_time:.2f}s, Average Prediction Time per Instance: {avg_pred_time:.5f}s")

# Generate and print confusion matrix
cm_knn = confusion_matrix(Y_Test, prediction_knn)
print(cm_knn)
```

K-Nearest Neighbors

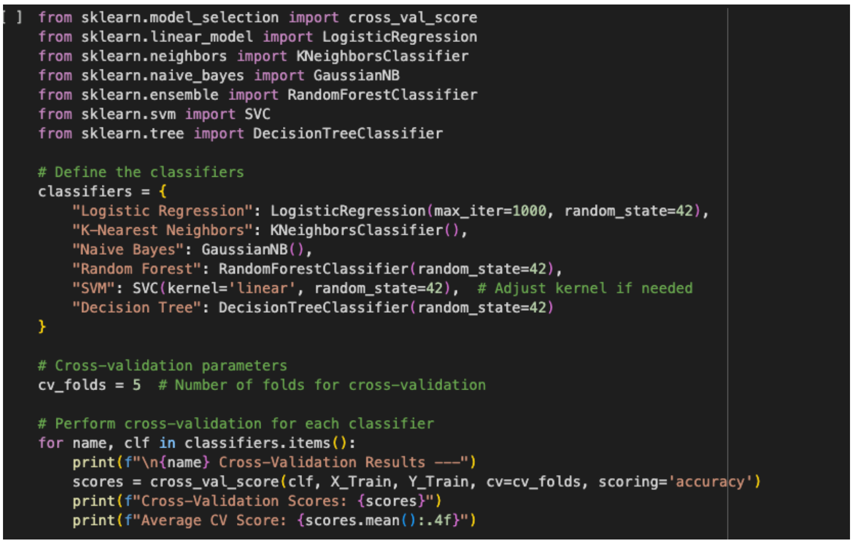


```python
# Train SVM Classifier
start_train = time.time()
svmclf = svm.SVC(random_state=42).fit(X_Train, Y_Train)  # Ensure consistent random state
train_time = time.time() - start_train

# Measure prediction time
start_test = time.time()
predictions_svm = svmclf.predict(X_Test)  # Store predictions in predictions_svm
test_time = time.time() - start_test

# Print classification report, accuracy, model size, training and prediction time
print(classification_report(Y_Test, predictions_svm, target_names=le.inverse_transform(svmclf.classes_), digits=3))
print(f"Accuracy: {svmclf.score(X_Test, Y_Test):.3f}")
print(f"Model size: {sys.getsizeof(pickle.dumps(svmclf))} bytes")
print(f"Training Time: {train_time:.2f}s, Prediction Time: {test_time:.2f}s")

# Generate and print confusion matrix
cm_svm = confusion_matrix(Y_Test, predictions_svm)
print(cm_svm)
```

Support Vector

```python
# Initialize the Naive Bayes model
nb_model = GaussianNB()

# Measure training time
start_train_time = time.time()
nb_model.fit(X_Train, Y_Train)
train_time = time.time() - start_train_time

# Measure testing time
start_test_time = time.time()
prediction_nb = nb_model.predict(X_Test)
test_time = time.time() - start_test_time

# Measure prediction time per instance
start_pred_time = time.time()
for instance in X_Test:
    nb_model.predict([instance])
avg_pred_time = (time.time() - start_pred_time) / len(X_Test)

# Print accuracy and classification report
print(classification_report(Y_Test, prediction_nb, target_names=le.inverse_transform(nb_model.classes_), digits=3))
print(f"Accuracy: {nb_model.score(X_Test, Y_Test)}")
print(f"Model size: {sys.getsizeof(pickle.dumps(nb_model))} bytes")
print(f"Training Time: {train_time:.2f}s, Testing Time: {test_time:.2f}s, Average Prediction Time per Instance: {avg_pred_time:.5f}s")

# Generate and print confusion matrix
cm_nb = confusion_matrix(Y_Test, prediction_nb)
print(cm_nb)
```

Naïve Bayes



```python
[ ]  from sklearn.model_selection import cross_val_score
     from sklearn.linear_model import LogisticRegression
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.naive_bayes import GaussianNB
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.svm import SVC
     from sklearn.tree import DecisionTreeClassifier

     # Define the classifiers
     classifiers = {
         "Logistic Regression": LogisticRegression(max_iter=1000, random_state=42),
         "K-Nearest Neighbors": KNeighborsClassifier(),
         "Naive Bayes": GaussianNB(),
         "Random Forest": RandomForestClassifier(random_state=42),
         "SVM": SVC(kernel='linear', random_state=42),  # Adjust kernel if needed
         "Decision Tree": DecisionTreeClassifier(random_state=42)
     }

     # Cross-validation parameters
     cv_folds = 5  # Number of folds for cross-validation

     # Perform cross-validation for each classifier
     for name, clf in classifiers.items():
         print(f"\n{name} Cross-Validation Results ---")
         scores = cross_val_score(clf, X_Train, Y_Train, cv=cv_folds, scoring='accuracy')
         print(f"Cross-Validation Scores: {scores}")
         print(f"Average CV Score: {scores.mean():.4f}")
```

Cross – Validation for base Classifiers.

Step 9. BoT_IoT - Base Classifiers Configuration.

7.2 Advanced Ensemble and Hybrid Stacking Models

CICIOT2023 - The Advanced Ensemble and Hybrid Stacking Models were configured with combining base learners and meta-learners. The tuned hybrid model followed DT-CART-driven by a maximum depth of 10-and XGBoost with 50 estimators and a maximum depth of 6-using logistic regression as a meta-learner and evaluated by 5-fold cross-validation. This was developed according to the ensemble: Hybrid Stacking Model: DT-CART max. Depth = 10, XGBoost tuned on log loss, and logistic regression-the meta-learner to combine predictions. Stacking Ensemble: XGBoost, Random Forest-50 estimators, max. Depth 6-decision trees max. Depth 6-logistic regression-stratified 3 fold for evaluation.

```python
# Define base models
base_learners = [
    ('dt', DecisionTreeClassifier(max_depth=10, random_state=42)),
    ('xgb', XGBClassifier(n_estimators=50, max_depth=6, eval_metric='mlogloss', random_state=42))
]

# Meta-learner
meta_learner = LogisticRegression(random_state=42)

# Stacking model
stacking_model = StackingClassifier(estimators=base_learners, final_estimator=meta_learner, cv=5)

# Fit the stacking model
print("Training the Tuned Hybrid Model...")
start_train_time = time.time()  # Start timer for training
stacking_model.fit(X_train, y_train)
end_train_time = time.time()  # End timer for training
training_time = end_train_time - start_train_time
print(f"Training Time: {training_time:.2f} seconds")

# Predict and measure prediction time
print("Predicting with the Tuned Hybrid Model...")
start_pred_time = time.time()  # Start timer for prediction
y_pred = stacking_model.predict(X_test)
end_pred_time = time.time()  # End timer for prediction
prediction_time = end_pred_time - start_pred_time
print(f"Prediction Time: {prediction_time:.2f} seconds")

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Tuned Hybrid Model Accuracy: {accuracy:.4f}")

# Classification report
classification_rep = classification_report(y_test, y_pred, target_names=label_encoder.classes_)
print("Classification Report for Tuned Hybrid Model:\n", classification_rep)

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Tuned Hybrid Model")
plt.show()
```

```python
# Define base learners
base_learners = [
    ('dt', DecisionTreeClassifier(max_depth=10, random_state=42)),
    ('xgb', XGBClassifier(n_estimators=50, max_depth=6, eval_metric='mlogloss', use_label_encoder=False, random_state=42))
]

# Define the meta-learner
meta_learner = LogisticRegression(max_iter=100, random_state=42)

# Define the stacking model
stacking_model = StackingClassifier(estimators=base_learners, final_estimator=meta_learner, cv=3)

# Prepare cross-validation
n_folds = 3
kf = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=42)

# Prepare lists to store metrics
accuracies = []
precisions = []
recalls = []
f1_scores = []
fold_runtimes = []

fold = 1

for train_index, test_index in kf.split(X, y):
    print(f"\n--- Fold {fold} ---")

    # Split the data for the current fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Track the start time
    start_time = time.time()

    # Train the stacking model
    stacking_model.fit(X_train, y_train)

    # Predict and evaluate
    y_pred = stacking_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Track end time and calculate runtime
    end_time = time.time()
    runtime = end_time - start_time
    fold_runtimes.append(runtime)

    # Store metrics
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)

    print(f"Fold {fold} Accuracy: {accuracy:.4f}")
    print(f"Fold {fold} Precision: {precision:.4f}")
    print(f"Fold {fold} Recall: {recall:.4f}")
    print(f"Fold {fold} F1 Score: {f1:.4f}")
    print(f"Fold {fold} Runtime: {runtime:.4f} seconds\n")

    fold += 1

# Calculate average metrics
avg_accuracy = np.mean(accuracies)
avg_precision = np.mean(precisions)
avg_recall = np.mean(recalls)
avg_f1 = np.mean(f1_scores)
avg_runtime = np.mean(fold_runtimes)

# Display cross-validation results
print("\n=== Cross-Validation Results for Tuned Hybrid Model ===")
print(f"Average Accuracy: {avg_accuracy:.4f}")
print(f"Average Precision: {avg_precision:.4f}")
print(f"Average Recall: {avg_recall:.4f}")
print(f"Average F1 Score: {avg_f1:.4f}")
print(f"Average Runtime per Fold: {avg_runtime:.4f} seconds")
```

Tuned Hybrid Model and Cross validation

```python
# Define base models
base_learners = [
    ('dt', DecisionTreeClassifier(max_depth=10, random_state=42)),
    ('xgb', XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', n_estimators=50, max_depth=6, random_state=42))
]

# Meta-learner
meta_learner = LogisticRegression(random_state=42)

# Stacking Model
stacking_model = StackingClassifier(estimators=base_learners, final_estimator=meta_learner, cv=5)
stacking_model.fit(X_train, y_train)

# Evaluate the hybrid model
y_pred = stacking_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Hybrid Stacking Model Accuracy: {accuracy:.4f}")
print("Classification Report for Hybrid Model:\n", classification_report(y_test, y_pred, target_names=label_encoder.classes_))

# Confusion Matrix Visualization
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Hybrid Model")
plt.show()
```

```python
# Set up Stratified K-Fold cross-validation
num_folds = 3
kf = StratifiedKFold(n_splits=num_folds, shuffle=True, random_state=42)

# Lists to store results across folds
accuracies = []
precisions = []
recalls = []
f1_scores = []
fold_runtimes = []

# Perform cross-validation
for fold, (train_index, test_index) in enumerate(kf.split(X, y), start=1):
    print(f"\n--- Fold {fold} ---")

    # Split data for the current fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Track the start time
    start_time = time.time()

    # Train the stacking model
    stacking_model.fit(X_train, y_train)

    # Predict and evaluate
    y_pred = stacking_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Track end time and calculate runtime
    end_time = time.time()
    runtime = end_time - start_time
    fold_runtimes.append(runtime)

    # Store metrics
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)

    # Display fold results
    print(f"Fold {fold} Accuracy: {accuracy:.4f}")
    print(f"Fold {fold} Precision: {precision:.4f}")
    print(f"Fold {fold} Recall: {recall:.4f}")
    print(f"Fold {fold} F1 Score: {f1:.4f}")
    print(f"Fold {fold} Runtime: {runtime:.4f} seconds\n")

# Average metrics across folds
print("\n--- Cross-Validation Results for Hybrid Model ---")
print(f"Average Accuracy: {np.mean(accuracies):.4f}")
print(f"Average Precision: {np.mean(precisions):.4f}")
print(f"Average Recall: {np.mean(recalls):.4f}")
print(f"Average F1 Score: {np.mean(f1_scores):.4f}")
print(f"Average Runtime per Fold: {np.mean(fold_runtimes):.4f} seconds")
```

Hybrid Stacking Model and Cross Validation

```python
# Step 1: Downsample the dataset
# Adjust the sample size to reduce computation time
sampled_df = balanced_df.sample(n=500000, random_state=42)

# Separate features and labels again
X = sampled_df.drop(columns=['label'])
y = sampled_df['label']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")

# Step 2: Reduce estimators and model complexity
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Define base models with reduced complexity
base_learners = [
    ('xgb', xgb.XGBClassifier(n_estimators=50, max_depth=6, eval_metric='mlogloss', use_label_encoder=False, random_state=42)),
    ('rf', RandomForestClassifier(n_estimators=50, max_depth=6, random_state=42)),
    ('dt', DecisionTreeClassifier(max_depth=6, random_state=42))
]

# Meta-learner
meta_learner = LogisticRegression(max_iter=100, random_state=42)

# Stacking ensemble
stacking_model = StackingClassifier(estimators=base_learners, final_estimator=meta_learner, cv=3)

# Step 3: Train the stacking model
stacking_model.fit(X_train, y_train)

# Step 4: Evaluate the model
y_pred = stacking_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Stacking Ensemble Accuracy: {accuracy:.4f}")

# Generate classification report
classification_rep = classification_report(y_test, y_pred, target_names=label_encoder.classes_)
print("Classification Report for Stacking Ensemble:\n", classification_rep)

#2.18s

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Stacking Ensemble")
plt.show()
```

```python
# Simplify base models to reduce training time
base_learners = [
    ('xgb', xgb.XGBClassifier(n_estimators=10, max_depth=3, eval_metric='mlogloss', random_state=42)),
    ('rf', RandomForestClassifier(n_estimators=10, max_depth=3, random_state=42)),
    ('dt', DecisionTreeClassifier(max_depth=3, random_state=42))
]

# Define the meta-learner
meta_learner = LogisticRegression(max_iter=100, random_state=42)

# Define the stacking model
stacking_model = StackingClassifier(estimators=base_learners, final_estimator=meta_learner, cv=3, n_jobs=-1)

# Set up StratifiedKFold for cross-validation
num_folds = 3  # Fewer folds for faster results
kf = StratifiedKFold(n_splits=num_folds, shuffle=True, random_state=42)

# Prepare lists to store metrics across folds
accuracies = []
precisions = []
recalls = []
f1_scores = []

fold = 1

# Perform cross-validation
for train_index, test_index in kf.split(X, y):
    print(f"\n--- Fold {fold} ---")
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train the stacking model on the current fold
    stacking_model.fit(X_train, y_train)

    # Predictions and evaluation on the test set for the current fold
    y_pred = stacking_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average="weighted")
    recall = recall_score(y_test, y_pred, average="weighted")
    f1 = f1_score(y_test, y_pred, average="weighted")

    # Append metrics to lists
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)

    # Print classification report for this fold
    print(f"Accuracy for Fold {fold}: {accuracy:.4f}")
    print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))

    fold += 1

# Calculate average metrics across folds
avg_accuracy = np.mean(accuracies)
avg_precision = np.mean(precisions)
avg_recall = np.mean(recalls)
avg_f1 = np.mean(f1_scores)

print("\n--- Cross-Validation Results ---")
print(f"Average Accuracy: {avg_accuracy:.4f}")
print(f"Average Precision: {avg_precision:.4f}")
print(f"Average Recall: {avg_recall:.4f}")
print(f"Average F1 Score: {avg_f1:.4f}")
```

Stacking Ensemble and Cross Validation

BoT_IoT- The hybrid model tuned combines a decision tree-CART with XGBoost. It includes the predictions from DT as extra features for XGBoost. Tuned parameters are max_depth=20 for DT and max_depth=6, n_estimators=200 for XGBoost. Hybrid Stacking Model combines DT and XGBoost. It takes predictions coming from DT and XGBoost as input features for the meta-learner: the logistic regression. The stacking ensemble, therefore, would combine XGBoost, RF, and DT with a meta-set and stack them onto LR. It uses predict_proba with 5-fold cross-validation to make the prediction more effective.

```python
# Set the best parameters for Decision Tree and XGBoost
best_dt_params = {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2}
best_xgb_params = {'subsample': 1.0, 'n_estimators': 200, 'max_depth': 6, 'learning_rate': 0.1}

# Define the function to train the hybrid model with tuned parameters
def train_tuned_hybrid_model(X_train, y_train):
    # Step 1: Train the Decision Tree model to generate additional features
    cart_model = DecisionTreeClassifier(random_state=42, **best_dt_params)
    cart_model.fit(X_train, y_train)

    # Generate CART predictions as additional features for XGBoost
    X_train_cart_pred = cart_model.predict(X_train).reshape(-1, 1)
    X_train_hybrid = np.concatenate((X_train, X_train_cart_pred), axis=1)

    # Step 2: Train the XGBoost model on the combined features
    xgb_model = XGBClassifier(random_state=42, tree_method='hist', **best_xgb_params)
    xgb_model.fit(X_train_hybrid, y_train)

    return xgb_model, cart_model

# Measure training time
start_training_time = time.time()
xgb_model_tuned, cart_model_tuned = train_tuned_hybrid_model(X_Train, Y_Train)
training_time = time.time() - start_training_time
print(f"Training Time for Tuned Hybrid Model: {training_time:.4f} seconds")

# Generate predictions for the test set
# Step 1: Generate CART predictions as additional features for the test set
X_test_cart_pred = cart_model_tuned.predict(X_Test).reshape(-1, 1)
X_test_hybrid = np.concatenate((X_Test, X_test_cart_pred), axis=1)

# Measure prediction time
start_prediction_time = time.time()
y_pred_hybrid = xgb_model_tuned.predict(X_test_hybrid)
prediction_time = time.time() - start_prediction_time
print(f"Prediction Time for Tuned Hybrid Model: {prediction_time:.4f} seconds")

# Step 3: Generate confusion matrix and classification report
conf_matrix = confusion_matrix(Y_Test, y_pred_hybrid)
class_report = classification_report(Y_Test, y_pred_hybrid, target_names=le.classes_)

# Display the classification report
print("Classification Report for Tuned Hybrid Model (DT-CART + XGBoost):")
print(class_report)

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=le.classes_, yticklabels=le.classes_)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Tuned Hybrid Model (DT-CART + XGBoost)")
plt.show()
```

```python
from sklearn.model_selection import cross_val_score
import numpy as np

# Define the function to cross-validate the tuned hybrid model
def cross_val_tuned_hybrid_model(X, Y, cv=5):
    # Use the tuned parameters for DT and XGBoost
    cart_model = DecisionTreeClassifier(random_state=42, **best_dt_params)
    xgb_model = XGBClassifier(random_state=42, tree_method='hist', **best_xgb_params)

    # Generate CART predictions as additional features for XGBoost
    cart_model.fit(X, Y)
    X_cart_pred = pd.DataFrame(cart_model.predict(X), columns=["Cart_Pred"])
    X_hybrid = pd.concat([pd.DataFrame(X), X_cart_pred], axis=1)

    # Cross-validate XGBoost on the hybrid features
    scores = cross_val_score(xgb_model, X_hybrid, Y, cv=cv, scoring="accuracy")
    print("Cross-Validation Scores for Tuned Hybrid Model (DT-CART + XGBoost):", scores)
    print("Mean CV Accuracy for Tuned Hybrid Model:", np.mean(scores))

# Run cross-validation for the Tuned Hybrid Model
cross_val_tuned_hybrid_model(X_Smote_Train, Y_Smote_Train)
```

Tuned Hybrid Model and Cross validation

```python
# Define base models
cart_model = DecisionTreeClassifier(random_state=42, max_depth=10)
xgb_model = XGBClassifier(n_estimators=100, max_depth=5, random_state=42, objective="multi:softmax")
meta_learner = LogisticRegression(max_iter=1000, random_state=42)

# Measure training time
start_time = time.time()

# Train base models and meta-learner
cart_model.fit(X_Smote_Train, Y_Smote_Train)
X_train_cart_pred = pd.DataFrame(cart_model.predict(X_Smote_Train), columns=["Cart_Pred"])

# Concatenate original features with CART predictions
X_train_hybrid = pd.concat([pd.DataFrame(X_Smote_Train), X_train_cart_pred], axis=1)

# Train XGBoost on the hybrid feature set
xgb_model.fit(X_train_hybrid, Y_Smote_Train)
X_train_xgb_pred = pd.DataFrame(xgb_model.predict(X_train_hybrid), columns=["XGB_Pred"])

# Train the meta-learner on XGBoost predictions
meta_learner.fit(X_train_xgb_pred, Y_Smote_Train)

training_time = time.time() - start_time

# Measure prediction time
start_time = time.time()

# Predictions on the test set
X_test_cart_pred = pd.DataFrame(cart_model.predict(X_Test), columns=["Cart_Pred"])
X_test_hybrid = pd.concat([pd.DataFrame(X_Test), X_test_cart_pred], axis=1)
X_test_xgb_pred = pd.DataFrame(xgb_model.predict(X_test_hybrid), columns=["XGB_Pred"])
y_pred_hybrid = meta_learner.predict(X_test_xgb_pred)

prediction_time = time.time() - start_time

# Evaluation
print("Classification Report for Hybrid Model (DT-CART + XGBoost + LR meta-learner):")
print(classification_report(Y_Test, y_pred_hybrid))
print(f"Training Time: {training_time:.2f} seconds")
print(f"Prediction Time: {prediction_time:.2f} seconds")

# Confusion Matrix
conf_matrix = confusion_matrix(Y_Test, y_pred_hybrid)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Hybrid Model (DT-CART + XGBoost + LR meta-learner)")
plt.show()
```

```python
from sklearn.model_selection import cross_val_score
import numpy as np

# Define cross-validation function for Hybrid Model
def cross_val_hybrid_model(X, Y, cv=5):
    cart_model = DecisionTreeClassifier(random_state=42, max_depth=10)
    xgb_model = XGBClassifier(n_estimators=100, max_depth=5, random_state=42, objective="multi:softmax")
    meta_learner = LogisticRegression(max_iter=1000, random_state=42)

    # Generate CART predictions as additional features for XGBoost
    cart_model.fit(X, Y)
    X_cart_pred = pd.DataFrame(cart_model.predict(X), columns=["Cart_Pred"])
    X_hybrid = pd.concat([pd.DataFrame(X), X_cart_pred], axis=1)

    # Generate XGBoost predictions to be used by the meta-learner
    xgb_model.fit(X_hybrid, Y)
    X_xgb_pred = pd.DataFrame(xgb_model.predict(X_hybrid), columns=["XGB_Pred"])

    # Cross-validation on meta-learner using XGBoost predictions
    scores = cross_val_score(meta_learner, X_xgb_pred, Y, cv=cv, scoring="accuracy")
    print("Cross-Validation Scores for Hybrid Model (DT-CART + XGBoost + LR meta-learner):", scores)
    print("Mean CV Accuracy for Hybrid Model:", np.mean(scores))

# Run cross-validation for Hybrid Model
cross_val_hybrid_model(X_Smote_Train, Y_Smote_Train)
```

Hybrid Stacking Model and Cross Validation

```python
# Define base models
base_models = [
    ('xgb', XGBClassifier(n_estimators=100, max_depth=5, random_state=42, objective="multi:softmax")),
    ('rf', RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)),
    ('dt', DecisionTreeClassifier(random_state=42, max_depth=10))
]

# Define meta-learner
meta_learner = LogisticRegression(max_iter=1000, random_state=42)

# Create the stacking ensemble
stacking_model = StackingClassifier(estimators=base_models, final_estimator=meta_learner, stack_method='predict_proba')

# Measure training time
start_time = time.time()
stacking_model.fit(X_Smote_Train, Y_Smote_Train)
training_time = time.time() - start_time

# Measure prediction time
start_time = time.time()
y_pred_stacking = stacking_model.predict(X_Test)
prediction_time = time.time() - start_time

# Evaluation
print("Classification Report for Stacking Ensemble (XGBoost, RF, DT, LR meta-learner):")
print(classification_report(Y_Test, y_pred_stacking, target_names=class_names))
print(f"Training Time: {training_time:.2f} seconds")
print(f"Prediction Time: {prediction_time:.2f} seconds")

# Confusion Matrix
conf_matrix = confusion_matrix(Y_Test, y_pred_stacking)
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix for Stacking Ensemble (XGBoost, RF, DT, LR meta-learner)")
plt.show()
```

```python
from sklearn.model_selection import cross_val_score
import numpy as np

# Define the stacking ensemble model
base_models = [
    ('xgb', XGBClassifier(n_estimators=100, max_depth=5, random_state=42, objective="multi:softmax")),
    ('rf', RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)),
    ('dt', DecisionTreeClassifier(random_state=42, max_depth=10))
]
meta_learner = LogisticRegression(max_iter=1000, random_state=42)
stacking_model = StackingClassifier(estimators=base_models, final_estimator=meta_learner, stack_method='predict_proba')

# Run cross-validation for Stacking Ensemble
cv_scores = cross_val_score(stacking_model, X_Smote_Train, Y_Smote_Train, cv=5, scoring="accuracy")
print("Cross-Validation Scores for Stacking Ensemble (XGBoost, RF, DT, LR meta-learner):", cv_scores)
print("Mean CV Accuracy for Stacking Ensemble:", np.mean(cv_scores))
```

Stacking Ensemble and Cross Validation

7.3 Federated Learning Approaches

CICIOT2023 – The Federated approaches include Basic Federated Learning in which the Decision Tree, Gradient Boosting, and AdaBoost classifiers were trained across multiple clients with the reduced parameters, using majority voting for predictions. Federated Ensemble with SMOTE applied SMOTE technique for balancing the dataset, and this is done before splitting it among clients, with similar models trained and predictions aggregated via majority voting. Federated Ensemble with Stacking and Majority Voting uses a stacking ensemble per client, combining the base models like DT, GB, AdaBoost with a Logistic Regression meta-learner and the final predictions were aggregated through majority voting. These methods address class imbalance, improve diversity, and enhance federated learning accuracy.

```python
# Assume your data is already prepared and split across clients (X_train_clients and y_train_clients)
# and your test data is split as X_test, y_test

# Track execution time
start_time = time.time()

# Define number of clients (reduced to 2 for faster testing)
num_clients = 2
client_models = []

# Train models for each client with reduced parameters
for i in range(num_clients):
    print(f"Training models for Client {i+1}...")

    # Initialize classifiers with reduced parameters
    dt_model = DecisionTreeClassifier(random_state=42, max_depth=5)  # Reduced depth
    gb_model = GradientBoostingClassifier(learning_rate=0.1, n_estimators=10)  # Fewer estimators
    ada_model = AdaBoostClassifier(n_estimators=10)  # Fewer estimators

    # Train each classifier on the client's data
    dt_model.fit(X_train_clients[i], y_train_clients[i])
    gb_model.fit(X_train_clients[i], y_train_clients[i])
    ada_model.fit(X_train_clients[i], y_train_clients[i])

    # Append the models for the client
    client_models.append([dt_model, gb_model, ada_model])

end_time = time.time()
print(f"Reduced Parameter Federated Ensemble Training Time: {end_time - start_time:.2f} seconds")

# Majority voting for predictions across clients
final_predictions = []
for i in range(len(X_test)):
    instance_votes = []
    for client in client_models:
        # Make predictions from each model for the test instance
        dt_pred = client[0].predict(X_test[i:i+1])[0]
        gb_pred = client[1].predict(X_test[i:i+1])[0]
        ada_pred = client[2].predict(X_test[i:i+1])[0]

        # Collect predictions for majority voting
        instance_votes.extend([dt_pred, gb_pred, ada_pred])

    # Majority vote for the instance
    final_prediction = max(set(instance_votes), key=instance_votes.count)
    final_predictions.append(final_prediction)

# Evaluation
print("\nClassification Report for Basic Federated Ensemble:")
print(classification_report(y_test, final_predictions))
print(f"Accuracy: {accuracy_score(y_test, final_predictions):.2f}")

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, final_predictions)
print("\nConfusion Matrix:")
print(conf_matrix)
```

Basic Federated Learning

```python
# Apply SMOTE to balance the training data before client split
smote = SMOTE(random_state=42, k_neighbors=1)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
print("Class distribution after SMOTE:", Counter(y_train_balanced))

# Define number of clients for federated learning
num_clients = 3
X_train_clients = np.array_split(X_train_balanced, num_clients)
y_train_clients = np.array_split(y_train_balanced, num_clients)

# Initialize client models list
client_models = []

# Train models for each client
for i in range(num_clients):
    print(f"Training models for Client {i + 1}...")

    # Base models for ensemble
    dt_model = DecisionTreeClassifier(max_depth=3, random_state=42)
    gb_model = GradientBoostingClassifier(n_estimators=10, learning_rate=0.1, random_state=42)
    ada_model = AdaBoostClassifier(n_estimators=10, random_state=42)

    # Train each model on the client's data
    dt_model.fit(X_train_clients[i], y_train_clients[i])
    gb_model.fit(X_train_clients[i], y_train_clients[i])
    ada_model.fit(X_train_clients[i], y_train_clients[i])

    # Append models to client list
    client_models.append((dt_model, gb_model, ada_model))

# Batch Prediction with Majority Voting
batch_size = 500  # Adjust based on your system's capability
final_predictions = []

for batch_start in range(0, len(X_test), batch_size):
    batch_end = min(batch_start + batch_size, len(X_test))
    batch_predictions = []

    # Predict in batch for each client
    for client in client_models:
        dt_pred = client[0].predict(X_test[batch_start:batch_end])
        gb_pred = client[1].predict(X_test[batch_start:batch_end])
        ada_pred = client[2].predict(X_test[batch_start:batch_end])

        # Collect predictions for the batch
        batch_predictions.append([dt_pred, gb_pred, ada_pred])

    # Aggregate predictions with majority voting
    for instance_idx in range(batch_end - batch_start):
        instance_votes = [batch_predictions[client_idx][model_idx][instance_idx]
                          for client_idx in range(num_clients)
                          for model_idx in range(3)]
        final_prediction = max(set(instance_votes), key=instance_votes.count)
        final_predictions.append(final_prediction)

# Evaluation
print("\nFederated Ensemble with SMOTE – Classifier Report:")
print(classification_report(y_test, final_predictions, target_names=class_names))
print(f"Accuracy: {accuracy_score(y_test, final_predictions):.2f}")

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, final_predictions)
print("\nConfusion Matrix for Federated Ensemble with SMOTE:")
print(conf_matrix)
```

Federated Ensemble with SMOTE

```python
# Apply SMOTE to balance the training data before client split
smote = SMOTE(random_state=42, k_neighbors=1)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
print("Class distribution after SMOTE:", Counter(y_train_balanced))

# Define number of clients for federated learning
num_clients = 3
X_train_clients = np.array_split(X_train_balanced, num_clients)
y_train_clients = np.array_split(y_train_balanced, num_clients)

# Initialize client models list
client_models = []

# Train models for each client
for i in range(num_clients):
    print(f"Training models for Client {i + 1}...")

    # Base models for stacking
    dt_model = DecisionTreeClassifier(max_depth=2, random_state=42)  # Reduced depth
    gb_model = GradientBoostingClassifier(n_estimators=3, learning_rate=0.3, random_state=42)  # Fewer estimators, higher learning rate
    ada_model = AdaBoostClassifier(n_estimators=3, random_state=42)  # Fewer estimators

    # Stacking model with reduced iterations for Logistic Regression
    stack_model = StackingClassifier(
        estimators=[('dt', dt_model), ('gb', gb_model), ('ada', ada_model)],
        final_estimator=LogisticRegression(max_iter=200),  # Reduced max_iter
        cv=3
    )

    # Train and store the model
    stack_model.fit(X_train_clients[i], y_train_clients[i])
    client_models.append(stack_model)

# Batch Prediction with Majority Voting
batch_size = 500  # Adjust based on your system's capability
final_predictions = []
for batch_start in range(0, len(X_test), batch_size):
    batch_end = min(batch_start + batch_size, len(X_test))
    batch_predictions = []

    # Predict in each batch for each client
    for client in client_models:
        batch_pred = client.predict(X_test[batch_start:batch_end])
        batch_predictions.append(batch_pred)

    # Aggregate predictions with majority voting
    for instance_idx in range(batch_end - batch_start):
        instance_votes = [
            batch_predictions[client_idx][instance_idx]
            for client_idx in range(len(batch_predictions))
        ]
        final_prediction = max(set(instance_votes), key=instance_votes.count)
        final_predictions.append(final_prediction)

# Evaluation
print("\nFederated Ensemble with Stacking and Majority Voting – Classifier Report:")
print(classification_report(y_test, final_predictions, target_names=class_names))
print(f"Accuracy: {accuracy_score(y_test, final_predictions):.2f}")

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, final_predictions)
print("\nConfusion Matrix for Federated Ensemble with Stacking and Majority Voting:")
print(conf_matrix)
```

Federated Ensemble with Stacking and Majority Voting

BoT_IoT – During the basic Federated learning setup, client data was distributed, and models (DT, Gradient Boosting, AdaBoost) were trained locally with majority voting for predictions. The SMOTE configuration applied class balancing on client data before model training and prediction aggregation. In the stacking and majority voting ensemble, a Logistic Regression meta-learner combined predictions from client models for improved performance.

```python
# Split the dataset into clients (for federated learning)
num_clients = 3
X_train_clients = np.array_split(X_Train, num_clients)
y_train_clients = np.array_split(Y_Train, num_clients)

# Initialize lists to store models
client_models_basic = []

# ---- Basic Federated Ensemble without Resampling ----
print("\nTraining Basic Federated Ensemble (No Resampling)...")
for i in range(num_clients):
    print(f"Training model for Client {i + 1} (Basic Federated Ensemble)...")

    # Define base models
    dt_model = DecisionTreeClassifier(max_depth=3, random_state=42)
    gb_model = GradientBoostingClassifier(n_estimators=10, learning_rate=0.1, random_state=42)
    ada_model = AdaBoostClassifier(n_estimators=10, random_state=42)

    # Define stacking model
    stack_model = StackingClassifier(
        estimators=[('dt', dt_model), ('gb', gb_model), ('ada', ada_model)],
        final_estimator=LogisticRegression(),
        cv=3
    )

    # Train model with client data
    stack_model.fit(X_train_clients[i], y_train_clients[i])
    client_models_basic.append(stack_model)

# ---- Evaluation for Basic Federated Ensemble ----
print("\nEvaluating Basic Federated Ensemble with Majority Voting...")
final_predictions_basic = []
for j in range(len(X_Test)):
    instance_votes = []
    for model in client_models_basic:
        pred = model.predict(X_Test[j].reshape(1, -1))[0]
        instance_votes.append(pred)
    final_prediction = max(set(instance_votes), key=instance_votes.count)  # Majority voting
    final_predictions_basic.append(final_prediction)

print("\nBasic Federated Ensemble - Classification Report:")
print(classification_report(Y_Test, final_predictions_basic, target_names=class_names))
print("Accuracy:", accuracy_score(Y_Test, final_predictions_basic))

# Confusion Matrix
conf_matrix_basic = confusion_matrix(Y_Test, final_predictions_basic)
print("\nConfusion Matrix for Basic Federated Ensemble:")
print(conf_matrix_basic)
```

Basic Federated Learning

```python
# Split the dataset into clients (for federated learning)
num_clients = 3
X_train_clients = np.array_split(X_Train, num_clients)
y_train_clients = np.array_split(Y_Train, num_clients)

# Initialize lists to store models
client_models_smote = []

# ---- Federated Ensemble with Conditional SMOTE ----
print("\nTraining Federated Ensemble with Conditional SMOTE or Random Oversampling...")
for i in range(num_clients):
    print(f"\nApplying SMOTE or Random Oversampling for Client {i + 1}...")

    # Check class distribution
    class_counts = Counter(y_train_clients[i])
    apply_smote = all(count >= 2 for count in class_counts.values())  # SMOTE requires at least 2 samples per class

    if apply_smote:
        print("Using SMOTE for balancing...")
        sampler = SMOTE(random_state=42, k_neighbors=1)
    else:
        print("Using Random Oversampling for balancing...")
        sampler = RandomOverSampler(random_state=42)

    # Apply the chosen sampler
    X_train_resampled, y_train_resampled = sampler.fit_resample(X_train_clients[i], y_train_clients[i])

    # Define base models
    dt_model = DecisionTreeClassifier(max_depth=3, random_state=42)
    gb_model = GradientBoostingClassifier(n_estimators=10, learning_rate=0.1, random_state=42)
    ada_model = AdaBoostClassifier(n_estimators=10, random_state=42)

    # Define stacking model
    stack_model = StackingClassifier(
        estimators=[('dt', dt_model), ('gb', gb_model), ('ada', ada_model)],
        final_estimator=LogisticRegression(),
        cv=3
    )

    # Train model with resampled data
    stack_model.fit(X_train_resampled, y_train_resampled)
    client_models_smote.append(stack_model)

# ---- Evaluation for Federated Ensemble with SMOTE ----
print("\nEvaluating Federated Ensemble with SMOTE and Majority Voting...")
final_predictions_smote = []
for j in range(len(X_Test)):
    instance_votes = []
    for model in client_models_smote:
        pred = model.predict(X_Test[j].reshape(1, -1))[0]
        instance_votes.append(pred)
    final_prediction = max(set(instance_votes), key=instance_votes.count)  # Majority voting
    final_predictions_smote.append(final_prediction)

print("\nFederated Ensemble with SMOTE (or Random Oversampling) - Classification Report:")
print(classification_report(Y_Test, final_predictions_smote, target_names=class_names))
print("Accuracy:", accuracy_score(Y_Test, final_predictions_smote))
```

Federated Ensemble with SMOTE

```python
# Define class names based on your specific dataset labels
class_names = ['Data_Exfiltration', 'DoS&DDoS', 'Keylogging', 'Normal', 'OS_Fingerprint', 'Service_Scan']

# Define number of clients for federated setup
num_clients = 3
X_train_clients = np.array_split(X_Train, num_clients)
y_train_clients = np.array_split(Y_Train, num_clients)

# Initialize client models list
client_models = []

# Train models for each client
for i in range(num_clients):
    print(f"Training models for Client {i + 1}...")

    # Base models for stacking
    dt_model = DecisionTreeClassifier(max_depth=3, random_state=42)
    gb_model = GradientBoostingClassifier(n_estimators=10, learning_rate=0.1, random_state=42)
    ada_model = AdaBoostClassifier(n_estimators=10, random_state=42)

    # Stacking model
    stack_model = StackingClassifier(
        estimators=[('dt', dt_model), ('gb', gb_model), ('ada', ada_model)],
        final_estimator=LogisticRegression(),
        cv=3
    )

    # Train and store the model for each client
    stack_model.fit(X_train_clients[i], y_train_clients[i])
    client_models.append(stack_model)

# Aggregate predictions with Majority Voting
final_predictions = []
for j in range(len(X_Test)):
    instance_votes = []
    for client in client_models:
        pred = client.predict(X_Test[j].reshape(1, -1))[0]
        instance_votes.append(pred)

    # Majority voting
    final_prediction = max(set(instance_votes), key=instance_votes.count)
    final_predictions.append(final_prediction)

# Evaluation
print("\nFederated Ensemble with Stacking and Majority Voting - Classifier Report:")
print(classification_report(Y_Test, final_predictions, target_names=class_names))
print(f"Accuracy: {accuracy_score(Y_Test, final_predictions):.2f}")

# Confusion Matrix
conf_matrix = confusion_matrix(Y_Test, final_predictions)
print("\nConfusion Matrix for Federated Ensemble with Stacking and Majority Voting:")
print(conf_matrix)
```

Federated Ensemble with Stacking and Majority Voting

8. Zero Trust Integration in Federated Learning

Step 1. Client ID Management and Access Control – Unique IDs has been assigned to the clients using the uuid4 function from the uuid library and the access permission were defined for the access control.

Step 2: Data Encryption and Decryption - Using the Fernet class, Data encryption was implemented from the cryptography library. Client data was encrypted before the transmission to simulate secure communication and decrypted it before training.

```python
import uuid

# Assign unique identities to each client
client_ids = [str(uuid.uuid4()) for _ in range(num_clients)]
client_permissions = {client_id: True for client_id in client_ids}  # Simplified access control

# Check permissions during model training
for i in range(num_clients):
    client_id = client_ids[i]
    if not client_permissions.get(client_id, False):
        print(f"Access denied for Client {client_id}")
        continue

    print(f"Training model for Client {client_id} with ID verification...")
    # Existing training code
```

Step 1. Client ID Management and Access Control

```python
from cryptography.fernet import Fernet

# Generate an encryption key
encryption_key = Fernet.generate_key()
cipher = Fernet(encryption_key)

# Simulate encrypting client data
encrypted_data = [cipher.encrypt(client_data.tobytes()) for client_data in X_train_clients]
print("Client data encrypted for secure transmission.")

# Decrypt data before training
decrypted_data = [cipher.decrypt(data) for data in encrypted_data]
X_train_clients = [np.frombuffer(data, dtype=X_train_clients[0].dtype).reshape(-1, X_train_clients[0].shape[1]) for data in decrypted_data]
print("Client data decrypted for training.")
```

Step 2: Data Encryption and Decryption

Step 3. Trust Score Initialization and Dynamic Adjustment – The Initial trust scores are set to 1.0 for all the clients and these trust scores will be changing dynamically and updated based on the client model accuracy during training.

Step 4. Secure Aggregation with Trust Filtering – These are then aggregated across trusted clients, and hence only trusted clients with a threshold ≥ 0.5 can contribute toward the aggregation process.

```python
client_trust_scores = {client_id: 1.0 for client_id in client_ids}  # Initialize all trust scores to 1.0

for i in range(num_clients):
    client_id = client_ids[i]
    print(f"Training model for Client {client_id}...")

    # Simulate trust score adjustments
    client_accuracy = accuracy_score(Y_Test, client_models_basic[i].predict(X_Test))
    client_trust_scores[client_id] *= client_accuracy  # Adjust trust score

    if client_trust_scores[client_id] < 0.5:
        print(f"Client {client_id} flagged as low trust. Investigating...")
        continue
```

Step 3. Trust Score Initialization and Dynamic Adjustment.

```python
final_predictions_secure = []

for j in range(len(X_Test)):
    instance_votes = []
    for i, model in enumerate(client_models_basic):
        client_id = client_ids[i]
        if client_trust_scores[client_id] >= 0.5:  # Only use trusted clients
            pred = model.predict(X_Test[j].reshape(1, -1))[0]
            instance_votes.append(pred)

    final_prediction = max(set(instance_votes), key=instance_votes.count)  # Majority voting
    final_predictions_secure.append(final_prediction)

print("Secure Aggregation with Trust Filtering completed.")
```

Step 4: Secure Aggregation with Trust Filtering

Step 5. Decryption Validation and Trust Logging – Decryption success is simulated and integrated into the trust score computation using weighted metrics (accuracy, consistency, data integrity). Logs are maintained for all client actions.

Step 6. Dataset Split into Clients – The dataset was split into three clients, with training features (X_train) and labels (Y_train) divided equally among them.

```python
# Check if decryption is successful (simulate success for all clients here)
try:
    # Attempt to decrypt data (assuming data is already encrypted earlier)
    decrypted_data = [cipher.decrypt(data) for data in encrypted_data]
    decryption_successful = True
except Exception as e:
    print("Decryption failed:", e)
    decryption_successful = False
```

```python
for i, model in enumerate(client_models_basic):
    client_id = client_ids[i]
    accuracy = accuracy_score(Y_Test, model.predict(X_Test))
    consistency = 1.0  # Assume consistency for simplicity
    data_integrity = 1.0 if decryption_successful else 0.0  # Incorporate encryption integrity

    # Weighted trust score update
    client_trust_scores[client_id] *= (0.6 * accuracy + 0.3 * consistency + 0.1 * data_integrity)

    if client_trust_scores[client_id] < 0.5:
        print(f"Client {client_id} flagged as low trust and excluded.")
```

```python
# Simulate decryption failure for a specific client
if i == 1:  # Assume client 1 has a decryption failure
    decryption_successful = False
else:
    decryption_successful = True
```

```python
import logging

logging.basicConfig(filename='zero_trust_simulation.log', level=logging.INFO)

for i, client_id in enumerate(client_ids):
    if client_trust_scores[client_id] < 0.5:
        logging.info(f"Client {client_id} excluded due to low trust. Trust Score: {client_trust_scores[client_id]}")
    else:
        logging.info(f"Client {client_id} included with Trust Score: {client_trust_scores[client_id]}")

logging.info("Final predictions secured with Trust Filtering.")
```

Step 5. Decryption Validation and Trust Logging

```python
[ ]  # Split the dataset into clients (assuming `num_clients` is defined)
     num_clients = 3  # Example: 3 clients
     X_train_clients = np.array_split(X_Train, num_clients)  # Split features
     Y_train_clients = np.array_split(Y_Train, num_clients)  # Split labels

     print(f"Dataset has been split into {num_clients} clients.")
```

Step 6. Dataset Split into Clients

Experiment 1 – Data Poisoning.

Step 7. Simulated Data Poisoning – For the client 1, a small portion of the training labels is altered randomly to simulate a data poisoning attack.

```python
# Simulate data poisoning for Client 1
poisoned_client = 0  # Select Client 1
Y_train_clients[poisoned_client][:10] = [random.choice(range(len(class_names))) for _ in range(10)]

print(f"Client {client_ids[poisoned_client]} has been poisoned.")
```

Step 7. Simulated Data Poisoning

Step 8. Recomputing Trust Scores – The poisoned client is assigned a trust score of 0, and any client with a trust score below 0.5 is flagged as low trust and excluded from aggregation.

```python
# Recompute trust scores
for i, model in enumerate(client_models_basic):
    client_id = client_ids[i]
    accuracy = accuracy_score(Y_Test, model.predict(X_Test)) if i != poisoned_client else 0.0  # Drop accuracy for poisoned client
    trust_score = accuracy  # Simplified for this step; can include data integrity if applicable
    client_trust_scores[client_id] = trust_score

    if trust_score < 0.5:
        print(f"{client_id} flagged as low trust and excluded.")
```

Step 8. Recomputing Trust Scores

Experiment 2 - failed decryption and adversarial predictions

Step 9. Simulating Failed Decryption for Client 2 – In experiment, the code assigns the simulated decryption failure for Client 2, by setting decryption_successful to False and data_integrity to 0.0 for the affected client.

Step 10. Simulating Adversarial Predictions for Client 3 - Random predictions are generated for Client 3 to mimic adversarial behavior, and the trust score for this client is calculated based on its prediction accuracy against the test labels.

```python
# Simulate failed decryption for Client 2
failed_client = 1
decryption_successful = False if client_id == client_ids[failed_client] else True
data_integrity = 1.0 if decryption_successful else 0.0
```

Step 9. Simulating Failed Decryption for Client 2.

```python
# Simulate adversarial predictions for Client 3
adversarial_client = 2
client_predictions = [random.choice(range(len(class_names))) for _ in range(len(Y_Test))]
trust_score = accuracy_score(Y_Test, client_predictions)  # Expect low trust
```

Step 10. Simulating Adversarial Predictions for client 3

Step 11. Recomputing Trust Scores After New Scenarios - Then the trust scores are recomputed by including the accuracy of adversarial predictions, data integrity in the case of failed decryption, weighted the adjustments for overall client trust. Clients with the trust scores less than the threshold limit of 0.7 are flagged and excluded

```python
# Recompute trust scores after simulating new scenarios
for i, model in enumerate(client_models_basic):
    client_id = client_ids[i]

    # Accuracy for adversarial client or normal client
    if i == adversarial_client:
        accuracy = trust_score  # Already computed for adversarial predictions
    else:
        accuracy = accuracy_score(Y_Test, model.predict(X_Test))

    # Data integrity for failed decryption
    data_integrity = 1.0 if i != failed_client else 0.0

    # Update trust score
    client_trust_scores[client_id] = 0.5 * accuracy + 0.5 * data_integrity

    if client_trust_scores[client_id] < 0.7:
        print(f"{client_id} flagged as low trust and excluded.")
```

Step 11. Recomputing Trust Scores After New Scenarios