Wireless Sensor Networks

Introduction:

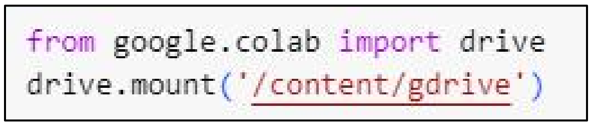
You are a data scientist working on a project to develop a machine learning model for classifying Denial of Service (DoS) attacks in Wireless Sensor Networks (WSN). The dataset provided for this task is named WSN-DS.

Scenario:

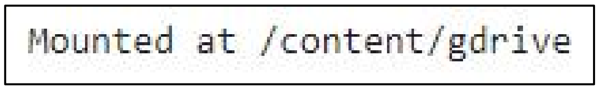
Imagine you are collaborating with a team of network security experts who aim to automate the detection of DoS attacks in WSN. They have collected data on node attributes and labels representing different types of attacks and normal behavior. Your team's goal is to build a reliable machine learning model that can accurately classify these attacks to assist in network security monitoring.

Step 1: Data Acquisition and Preparation

- **Load the provided dataset into the Google Colab environment using Python libraries.**

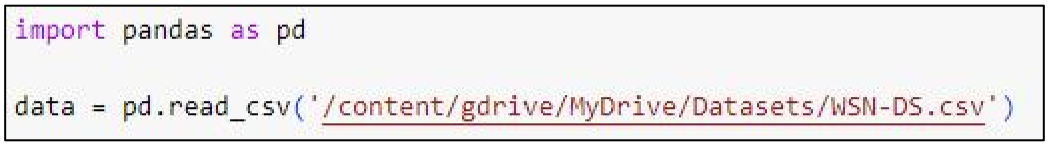


```
from google.colab import drive
drive.mount('/content/gdrive')
```



```
Mounted at /content/gdrive
```

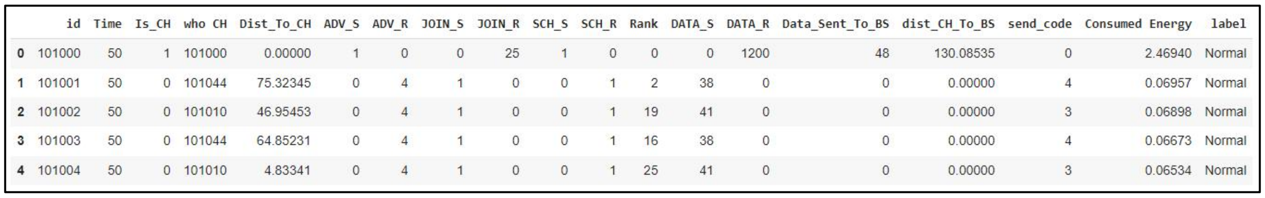
First, we need to load the dataset into the Google Colab environment. You can upload the dataset file to your Google Drive and then load it using Python libraries such as `pandas`.



```
import pandas as pd

data = pd.read_csv('/content/gdrive/MyDrive/Datasets/WSN-DS.csv')
```

View the first few rows:

```
data.head()
```



	id	Time	Is_CH	who CH	Dist_To_CH	ADV_S	ADV_R	JOIN_S	JOIN_R	SCH_S	SCH_R	Rank	DATA_S	DATA_R	Data_Sent_To_BS	dist_CH_To_BS	send_code	Consumed Energy	label
0	101000	50	1	101000	0.00000	1	0	0	25	1	0	0	0	1200	48	130.08535	0	2.46940	Normal
1	101001	50	0	101044	75.32345	0	4	1	0	0	1	2	38	0	0	0.00000	4	0.06957	Normal
2	101002	50	0	101010	46.95453	0	4	1	0	0	1	19	41	0	0	0.00000	3	0.06898	Normal
3	101003	50	0	101044	64.85231	0	4	1	0	0	1	16	38	0	0	0.00000	4	0.06673	Normal
4	101004	50	0	101010	4.83341	0	4	1	0	0	1	25	41	0	0	0.00000	3	0.06534	Normal

- **Explore the dataset to understand its structure, statistical properties, and potential challenges.**

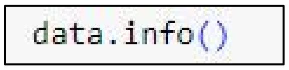
To understand the structure, statistical properties, and potential challenges of the dataset, we can perform some exploratory data analysis (EDA).

Check the number of rows and columns:

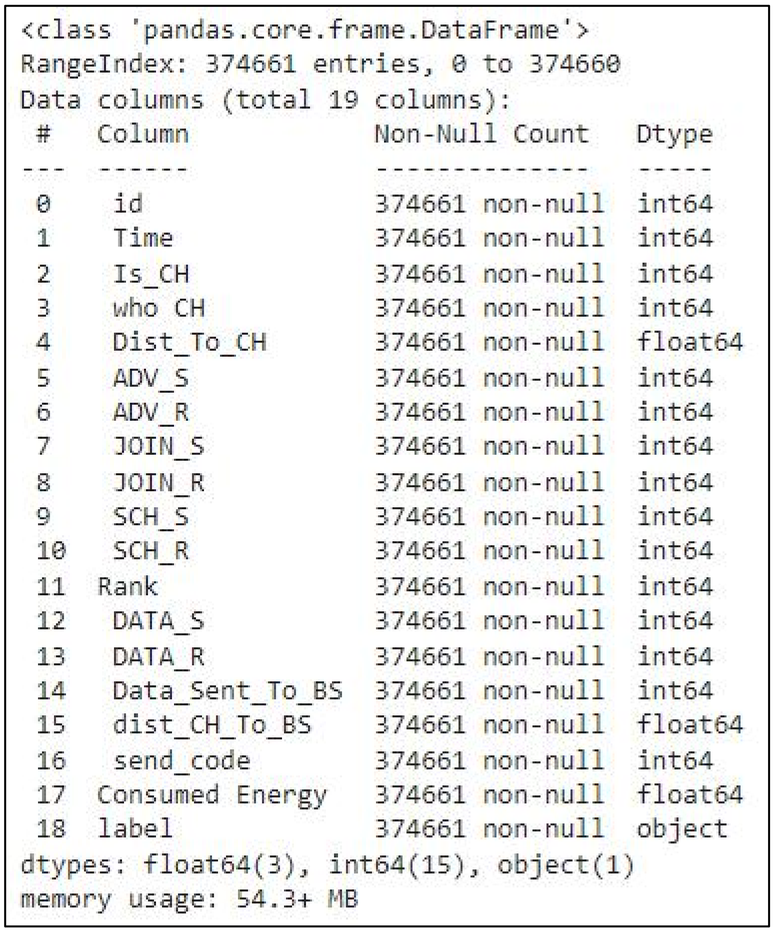
```
print(f"Number of Rows: {data.shape[0]}")
print(f"Number of Columns: {data.shape[1]}")
```

```
Number of Rows: 374661
Number of Columns: 19
```

Get information about each column:



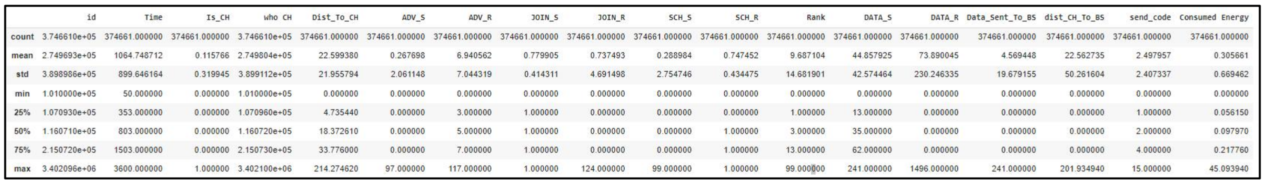
```
data.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 374661 entries, 0 to 374660
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0    id               374661 non-null  int64
 1    Time             374661 non-null  int64
 2    Is_CH            374661 non-null  int64
 3    who CH           374661 non-null  int64
 4    Dist_To_CH       374661 non-null  float64
 5    ADV_S            374661 non-null  int64
 6    ADV_R            374661 non-null  int64
 7    JOIN_S           374661 non-null  int64
 8    JOIN_R           374661 non-null  int64
 9    SCH_S            374661 non-null  int64
 10   SCH_R            374661 non-null  int64
 11  Rank              374661 non-null  int64
 12   DATA_S           374661 non-null  int64
 13   DATA_R           374661 non-null  int64
 14   Data_Sent_To_BS  374661 non-null  int64
 15   dist_CH_To_BS    374661 non-null  float64
 16   send_code        374661 non-null  int64
 17  Consumed Energy   374661 non-null  float64
 18  label             374661 non-null  object
dtypes: float64(3), int64(15), object(1)
memory usage: 54.3+ MB
```

Get descriptive statistics:

```
data.describe()
```



	id	Time	Is_CH	who CH	Dist_To_CH	ADV_S	ADV_R	JOIN_S	JOIN_R	SCH_S	SCH_R	Rank	DATA_S	DATA_R	Data_Sent_To_BS	dist_CH_To_BS	send_code	Consumed Energy
count	3.746610e+05	374661.000000	374661.000000	3.746610e+05	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000	374661.000000
mean	2.749693e+05	1064.748712	0.115766	2.749804e+05	22.599380	0.267698	6.940562	0.779905	0.737493	0.288984	0.747452	9.687104	44.857925	73.890045	4.569448	22.562735	2.497957	0.305661
std	3.898986e+05	899.646164	0.319945	3.899112e+05	21.955794	2.061148	7.044319	0.414311	4.691498	2.754746	0.434475	14.681901	42.574464	230.246335	19.679155	50.261604	2.407337	0.669462
min	1.010000e+05	50.000000	0.000000	1.010000e+05	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.070930e+05	353.000000	0.000000	1.070960e+05	4.735440	0.000000	3.000000	1.000000	0.000000	0.000000	0.000000	1.000000	13.000000	0.000000	0.000000	0.000000	1.000000	0.056150
50%	1.160710e+05	803.000000	0.000000	1.160720e+05	18.372610	0.000000	5.000000	1.000000	0.000000	0.000000	1.000000	3.000000	35.000000	0.000000	0.000000	0.000000	2.000000	0.097970
75%	2.150720e+05	1503.000000	0.000000	2.150730e+05	33.776000	0.000000	7.000000	1.000000	0.000000	0.000000	1.000000	13.000000	62.000000	0.000000	0.000000	0.000000	4.000000	0.217760
max	3.402096e+06	3600.000000	1.000000	3.402100e+06	214.274620	97.000000	117.000000	1.000000	124.000000	99.000000	1.000000	99.000000	241.000000	1496.000000	241.000000	201.934940	15.000000	45.093940

Display class distributions:

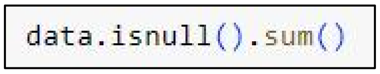
```
print("Class Distribution:\n", data.iloc[:, -1].value_counts())
```

```
Class Distribution:
 label
Normal        340066
Grayhole       14596
Blackhole      10049
TDMA            6638
Flooding        3312
Name: count, dtype: int64
```

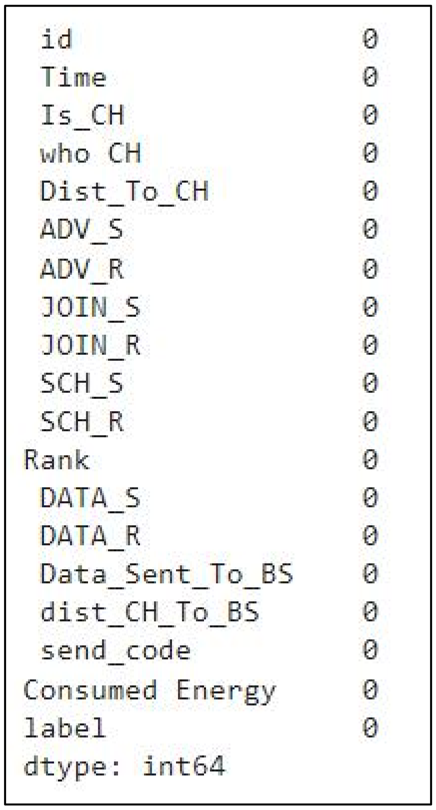
- **Preprocess the data by handling missing values, outliers, or inconsistencies to ensure quality data for analysis.**

Preprocessing involves handling missing values, outliers, or inconsistencies. Here, we assume there are no missing values based on the dataset description, but let's confirm and handle outliers if necessary.

Check for missing values:

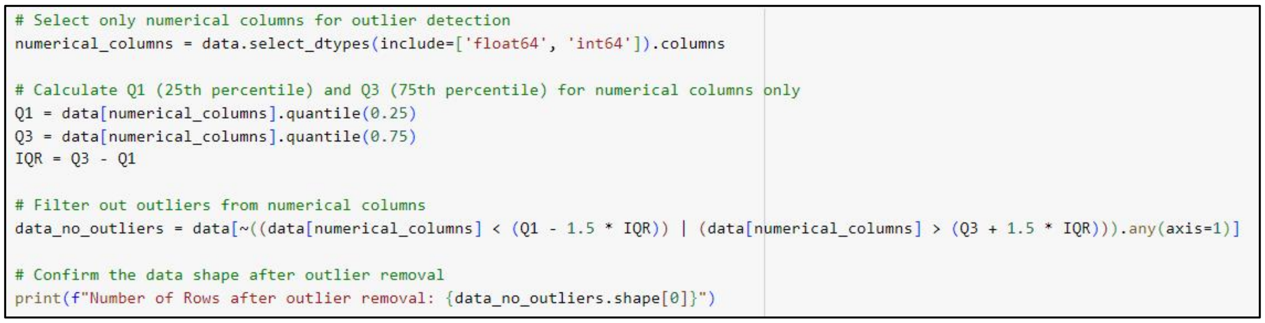


```
data.isnull().sum()
```

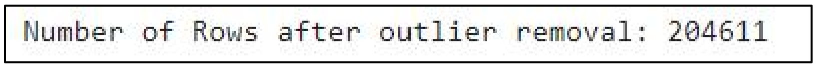


```
 id                    0
 Time                  0
 Is_CH                 0
 who CH                0
 Dist_To_CH            0
 ADV_S                 0
 ADV_R                 0
 JOIN_S                0
 JOIN_R                0
 SCH_S                 0
 SCH_R                 0
Rank                   0
 DATA_S                0
 DATA_R                0
 Data_Sent_To_BS       0
 dist_CH_To_BS         0
 send_code             0
Consumed Energy        0
label                  0
dtype: int64
```

Remove Outliers:



```
# Select only numerical columns for outlier detection
numerical_columns = data.select_dtypes(include=['float64', 'int64']).columns

# Calculate Q1 (25th percentile) and Q3 (75th percentile) for numerical columns only
Q1 = data[numerical_columns].quantile(0.25)
Q3 = data[numerical_columns].quantile(0.75)
IQR = Q3 - Q1

# Filter out outliers from numerical columns
data_no_outliers = data[~((data[numerical_columns] < (Q1 - 1.5 * IQR)) | (data[numerical_columns] > (Q3 + 1.5 * IQR))).any(axis=1)]

# Confirm the data shape after outlier removal
print(f"Number of Rows after outlier removal: {data_no_outliers.shape[0]}")
```

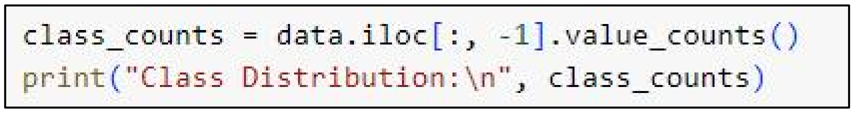


```
Number of Rows after outlier removal: 204611
```

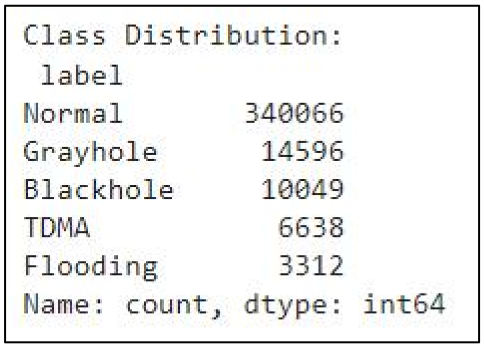
- **Analyze the distribution of classes within the dataset and discuss any imbalances or challenges that may arise during classification.**

It's crucial to analyze the distribution of classes to identify any imbalances.

Calculate Class Distribution:

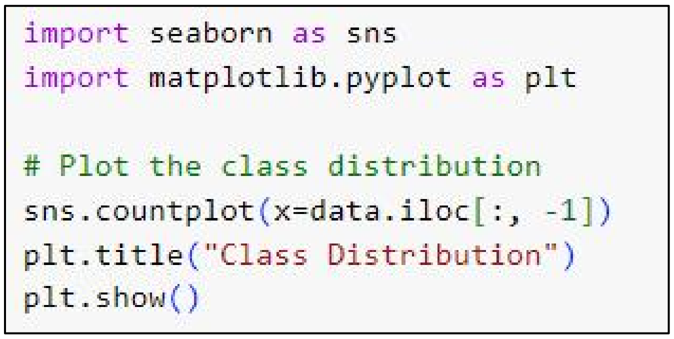


```
class_counts = data.iloc[:, -1].value_counts()
print("Class Distribution:\n", class_counts)
```

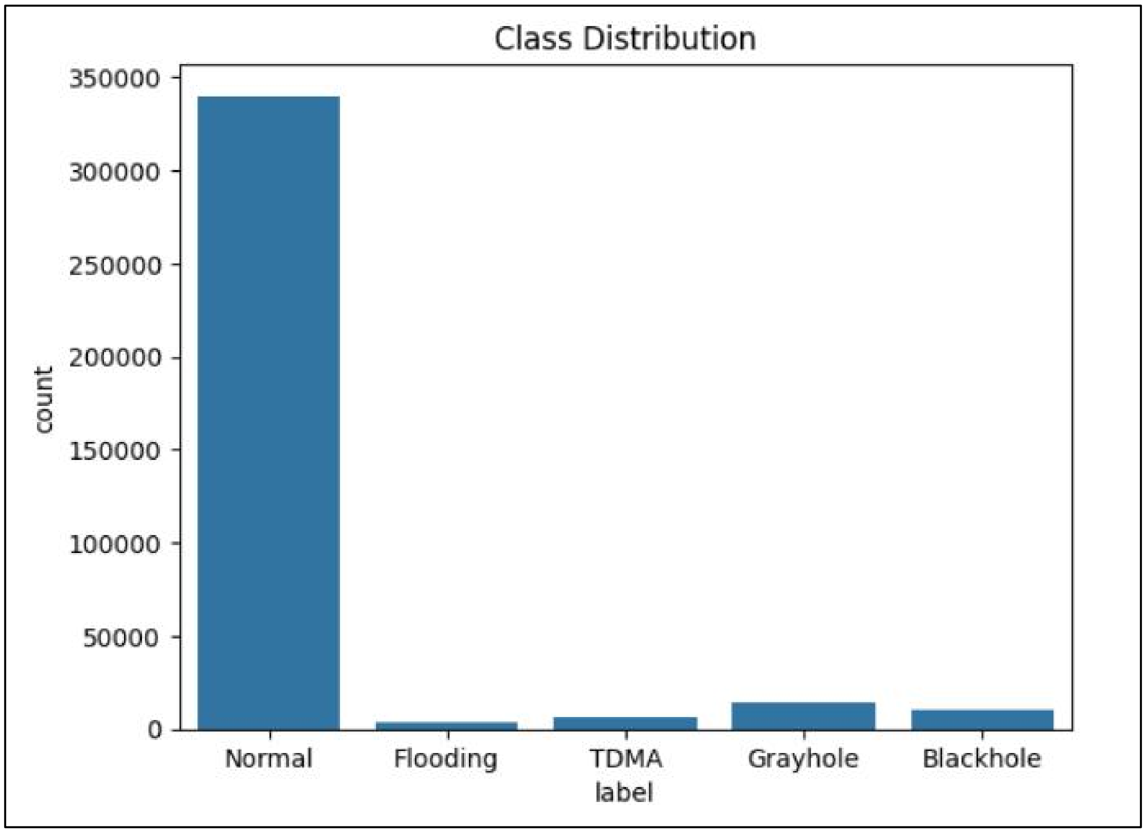


```
Class Distribution:
 label
Normal       340066
Grayhole      14596
Blackhole     10049
TDMA           6638
Flooding       3312
Name: count, dtype: int64
```

Plot the Class Distribution:



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

# Plot the class distribution
sns.countplot(x=data.iloc[:, -1])
plt.title("Class Distribution")
plt.show()
```



- **Balance the dataset if class imbalances are identified to prevent biases in model training.**

If class imbalances are identified, we need to balance the dataset to prevent biases in model training. We can use techniques like over sampling, under sampling, or SMOTE (Synthetic Minority Over-sampling Technique).

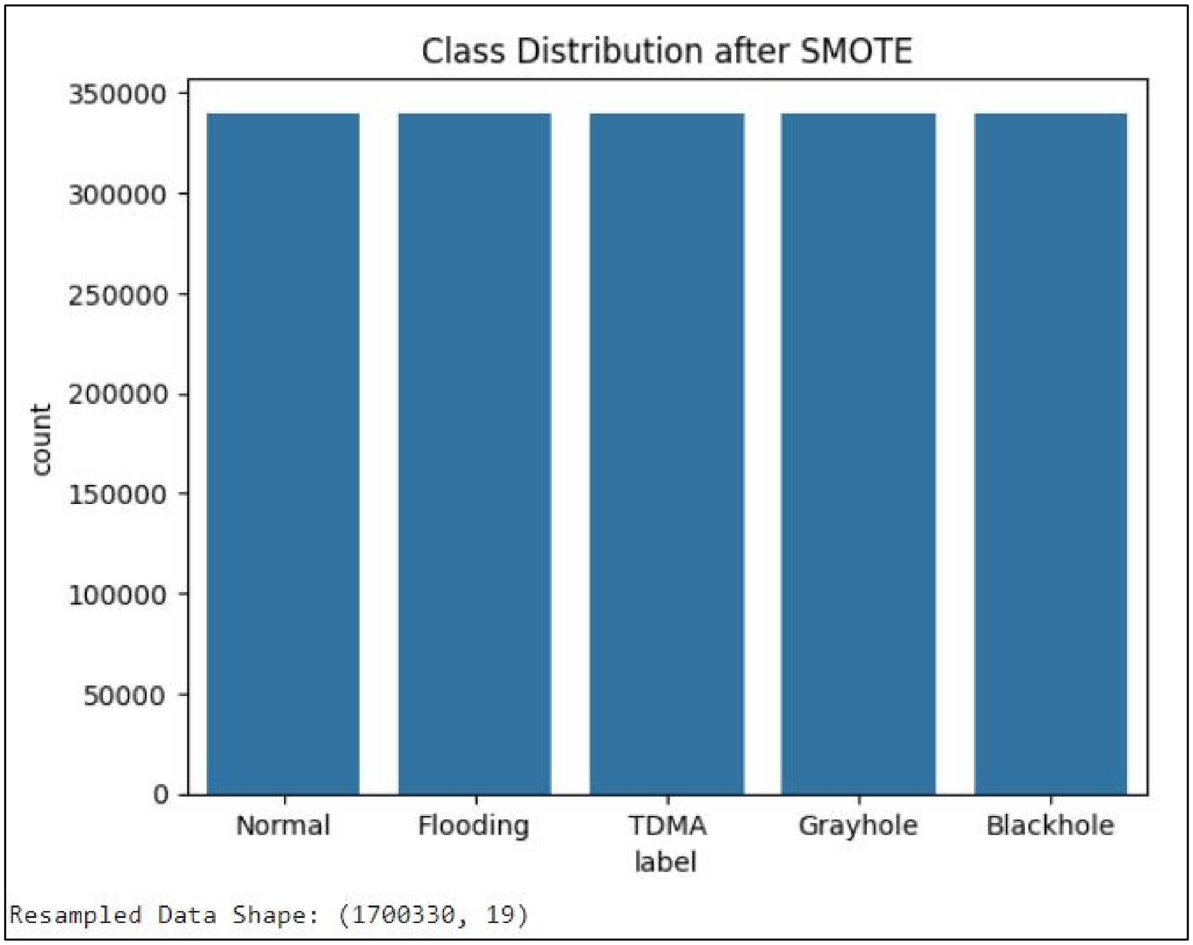
Identify Imbalances Ratio:

```
majority_class_count = class_counts.max()
minority_class_count = class_counts.min()
imbalance_ratio = majority_class_count / minority_class_count
print(imbalance_ratio)
```

```
102.67693236714976
```

Balance the Dataset:

```
from imblearn.over_sampling import SMOTE

# Separate features and labels
X = data.iloc[:, :-1]
y = data.iloc[:, -1]

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

# Verify the class distribution after resampling
sns.countplot(x=y_resampled)
plt.title("Class Distribution after SMOTE")
plt.show()

# Combine the resampled features and labels
data_resampled = pd.concat([X_resampled, y_resampled], axis=1)
print("Resampled Data Shape:", data_resampled.shape)
```



```
Resampled Data Shape: (1700330, 19)
```

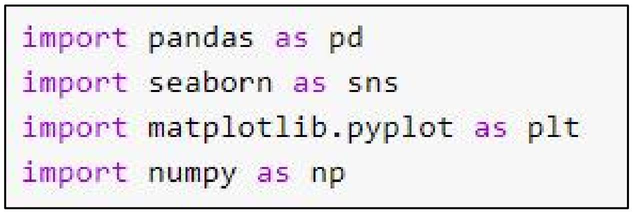
Now the dataset is preprocessed and balanced, and you can proceed to the next steps in your machine learning workflow, such as feature engineering, model selection, training, and evaluation.

Step 2: Exploratory Data Analysis and Visualization

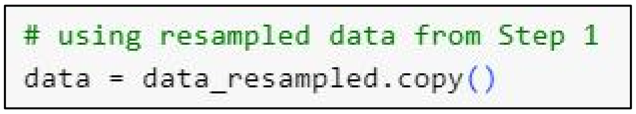
- **Perform comprehensive exploratory data analysis to gain insights into the WSN dataset.**

To perform comprehensive exploratory data analysis (EDA) and visualization, follow these steps:

Import necessary libraries:



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

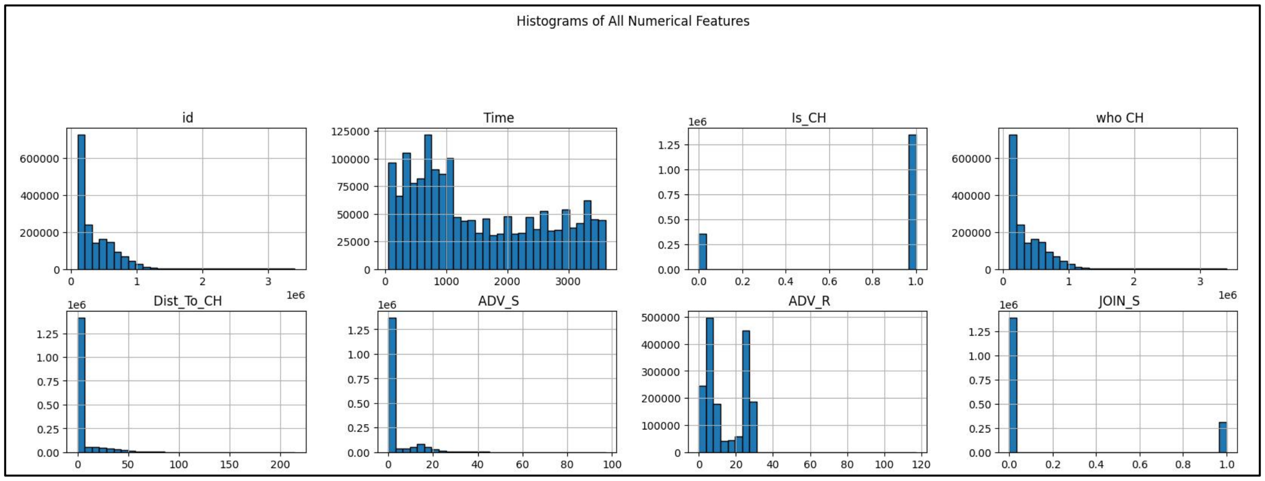


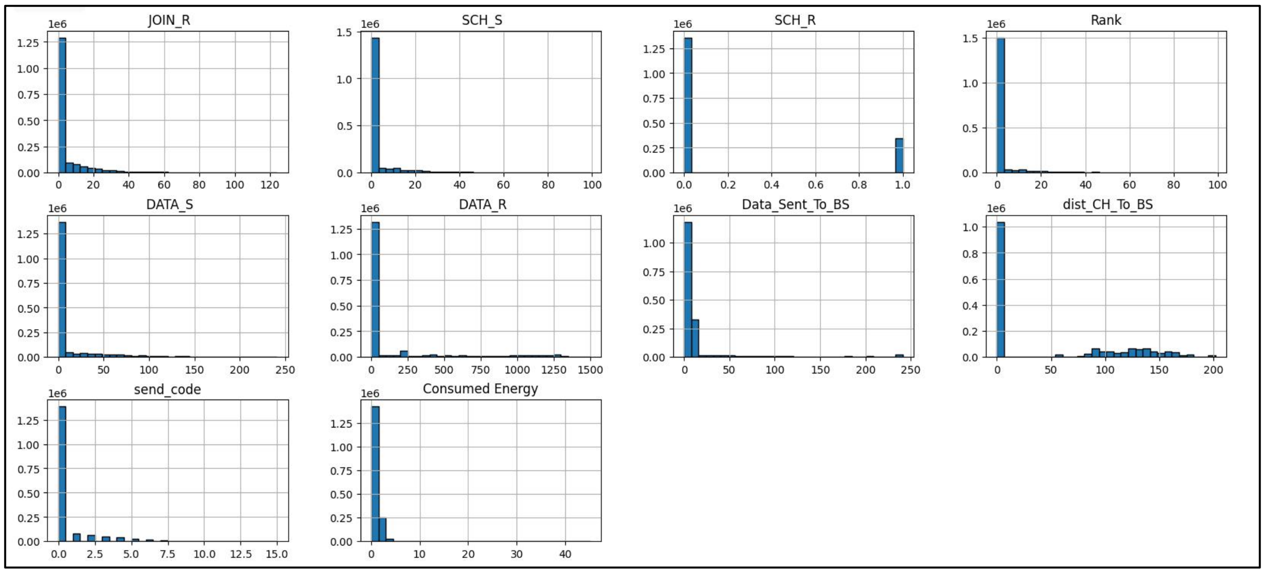
```
# using resampled data from Step 1
data = data_resampled.copy()
```

- **Utilize data visualization techniques such as histograms, pie chart, box plots, pair plots, and correlation matrices to visualize the distribution of features and understand the relationships between them.**

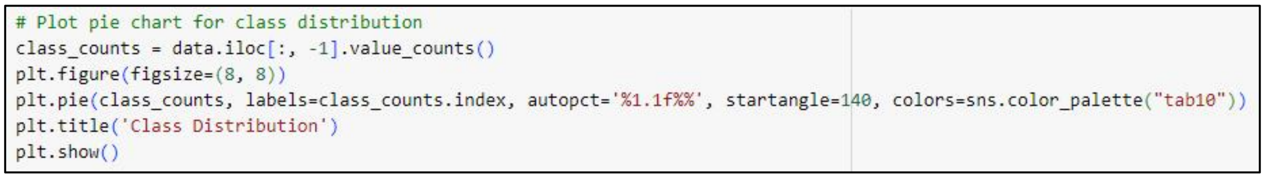
Histogram:

Visualize the distribution of numerical features.

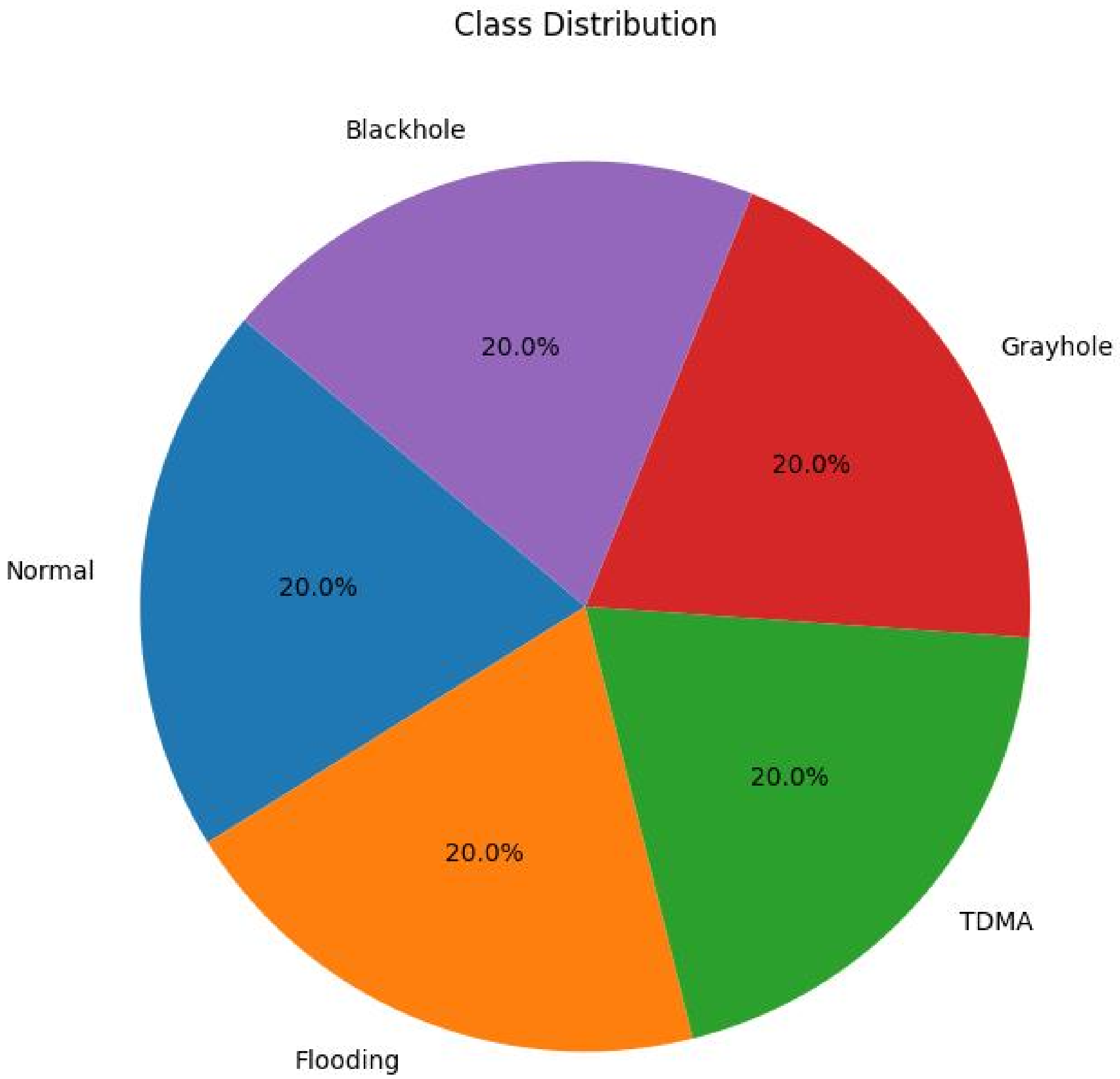




Pie Chart:
Visualize the distribution of classes.

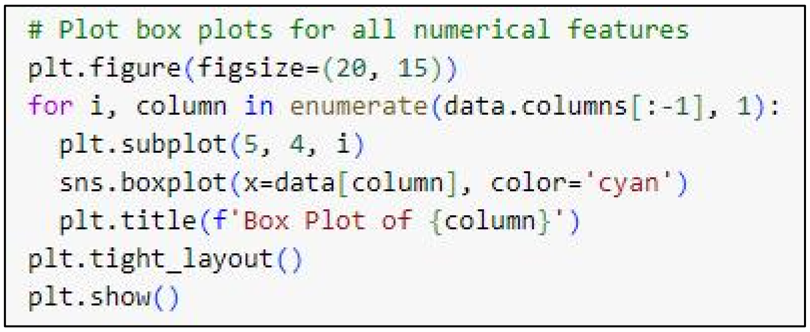


```
# Plot pie chart for class distribution
class_counts = data.iloc[:, -1].value_counts()
plt.figure(figsize=(8, 8))
plt.pie(class_counts, labels=class_counts.index, autopct='%1.1f%%', startangle=140, colors=sns.color_palette("tab10"))
plt.title('Class Distribution')
plt.show()
```

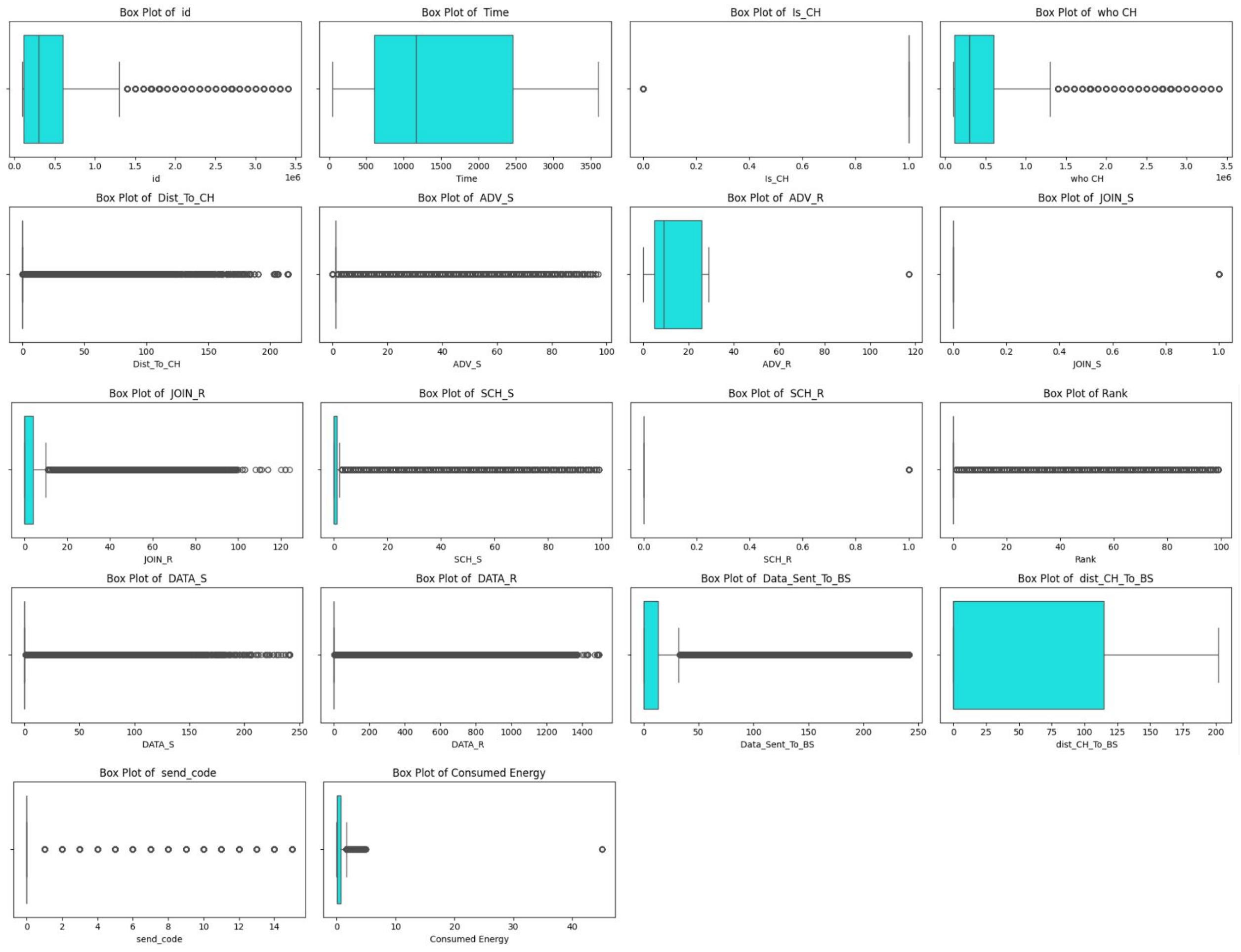


Box Plots:

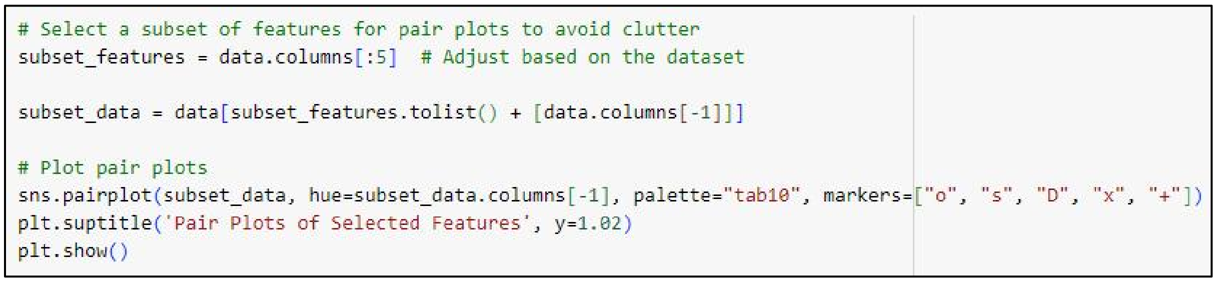
Visualize the spread and potential outliers of numerical features.



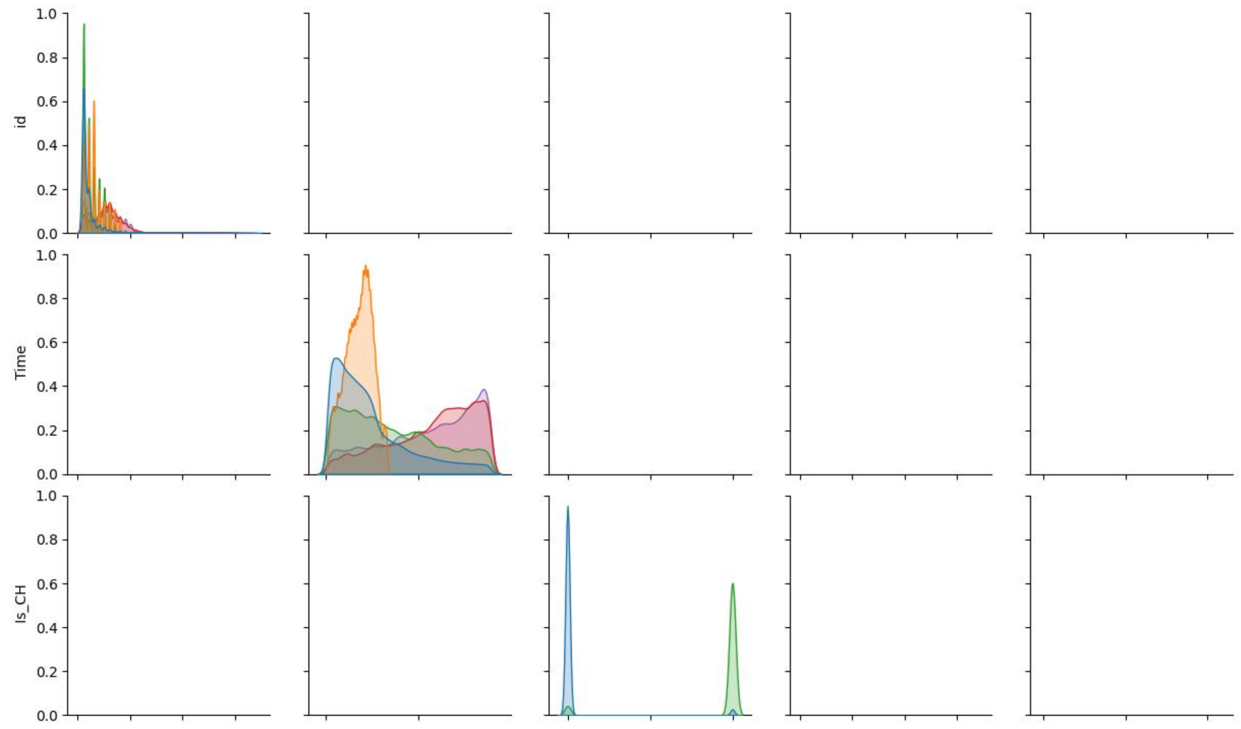
```
# Plot box plots for all numerical features
plt.figure(figsize=(20, 15))
for i, column in enumerate(data.columns[:-1], 1):
  plt.subplot(5, 4, i)
  sns.boxplot(x=data[column], color='cyan')
  plt.title(f'Box Plot of {column}')
plt.tight_layout()
plt.show()
```

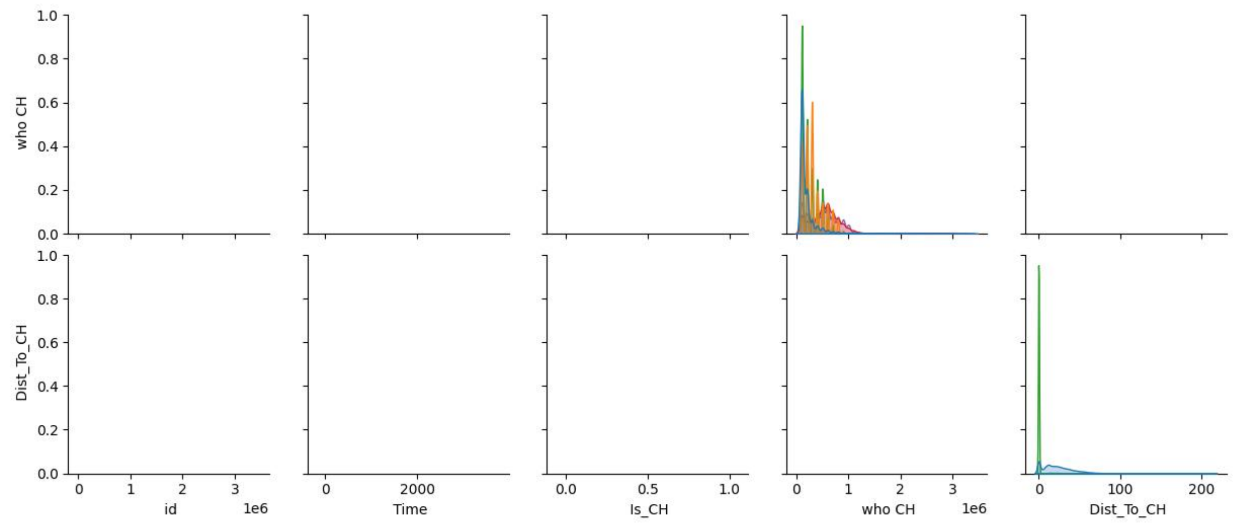


Pair Plots:
Visualize relationships between features and classes.

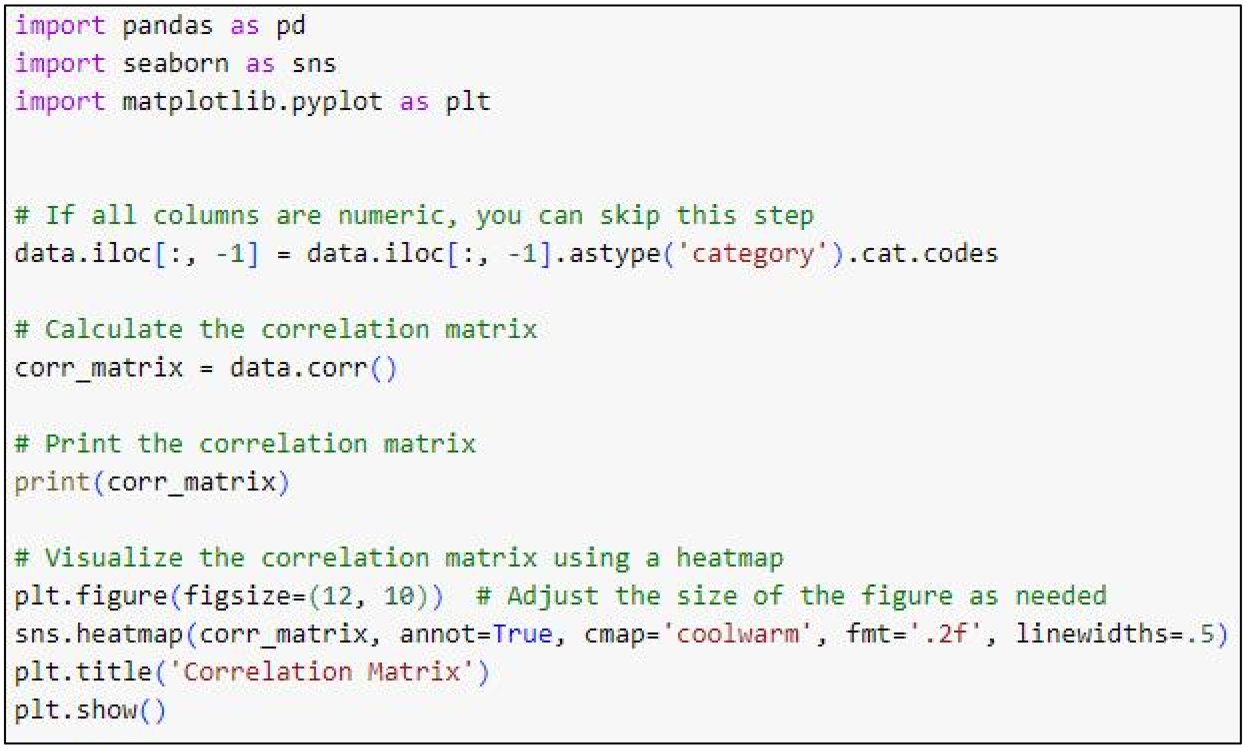


```
# Select a subset of features for pair plots to avoid clutter
subset_features = data.columns[:5]  # Adjust based on the dataset

subset_data = data[subset_features.tolist() + [data.columns[-1]]]

# Plot pair plots
sns.pairplot(subset_data, hue=subset_data.columns[-1], palette="tab10", markers=["o", "s", "D", "x", "+"])
plt.suptitle('Pair Plots of Selected Features', y=1.02)
plt.show()
```

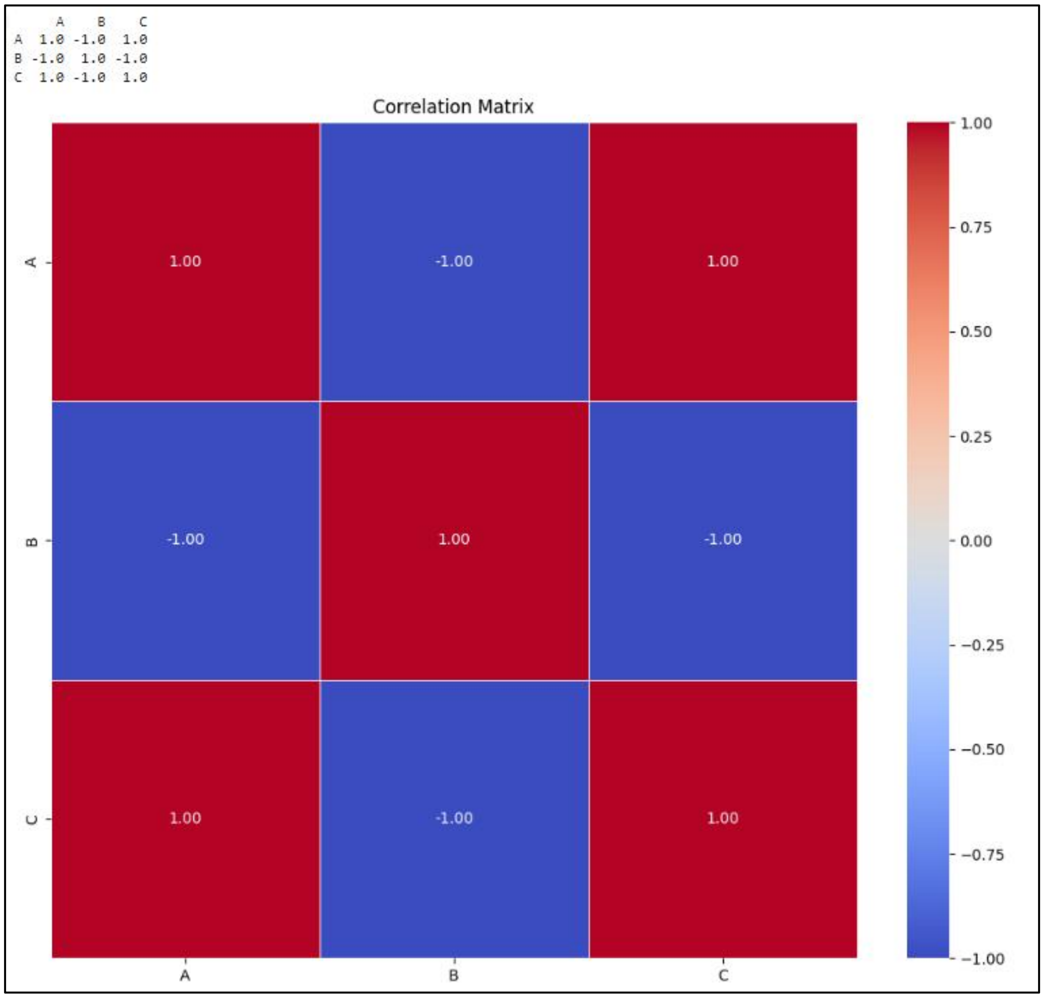




Correlation Matrices:



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt



# If all columns are numeric, you can skip this step
data.iloc[:, -1] = data.iloc[:, -1].astype('category').cat.codes

# Calculate the correlation matrix
corr_matrix = data.corr()

# Print the correlation matrix
print(corr_matrix)

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(12, 10))  # Adjust the size of the figure as needed
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=.5)
plt.title('Correlation Matrix')
plt.show()
```



```
     A    B    C
A  1.0 -1.0  1.0
B -1.0  1.0 -1.0
C  1.0 -1.0  1.0
```

- **Identify any patterns or trends in the data that could aid in classification tasks.**

Using these visualizations, look for patterns or trends such as:

- **High correlations** between certain features.
- **Distinct clusters** in pair plots indicating separability between classes.
- **Outliers** in box plots that might affect model performance.
- **Class distribution** balance or imbalance from pie charts.

These EDA steps provide valuable insights into the WSN-DS dataset, guiding feature selection and model development for classifying DoS attacks,

Step 3: Feature Engineering

- **Extract relevant features from the dataset.**

Start by identifying and extracting the relevant features from the dataset. For this dataset, all 18 columns are considered features, and the last column is the label.

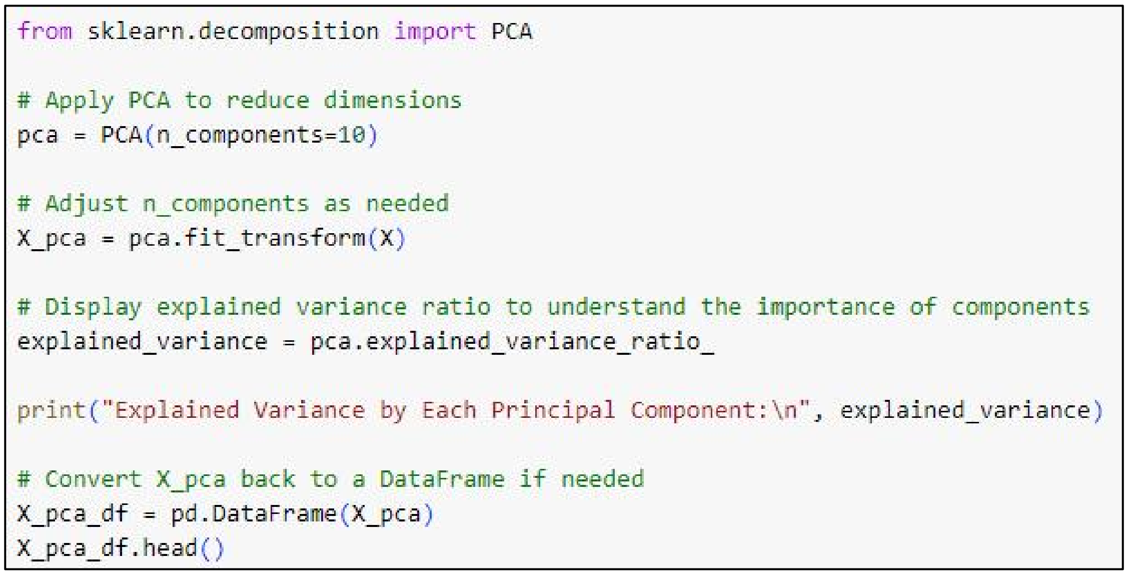
Extract Relevant Features:

```
# Separate features (X) and labels (y)
X = data.iloc[:, :-1]
y = data.iloc[:, -1]

# Display the first few rows of features and labels
X.head()
y.head()
```

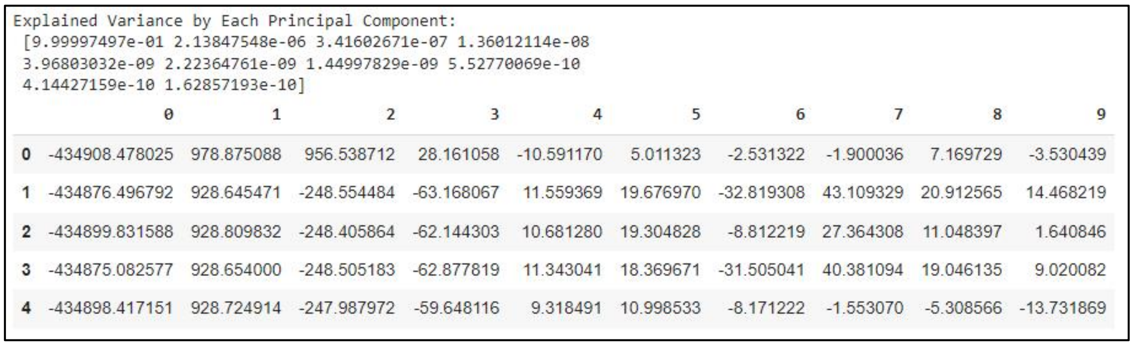
```
0    Normal
1    Normal
2    Normal
3    Normal
4    Normal
Name: label, dtype: object
```

- **Perform dimensionality reduction techniques if needed to reduce computational complexity.**

If the dataset has high dimensionality, dimensionality reduction techniques like Principal Component Analysis (PCA) can be used to reduce computational complexity. This step is optional and depends on the specific dataset and computational resources.



```python
from sklearn.decomposition import PCA

# Apply PCA to reduce dimensions
pca = PCA(n_components=10)

# Adjust n_components as needed
X_pca = pca.fit_transform(X)

# Display explained variance ratio to understand the importance of components
explained_variance = pca.explained_variance_ratio_

print("Explained Variance by Each Principal Component:\n", explained_variance)

# Convert X_pca back to a DataFrame if needed
X_pca_df = pd.DataFrame(X_pca)
X_pca_df.head()
```

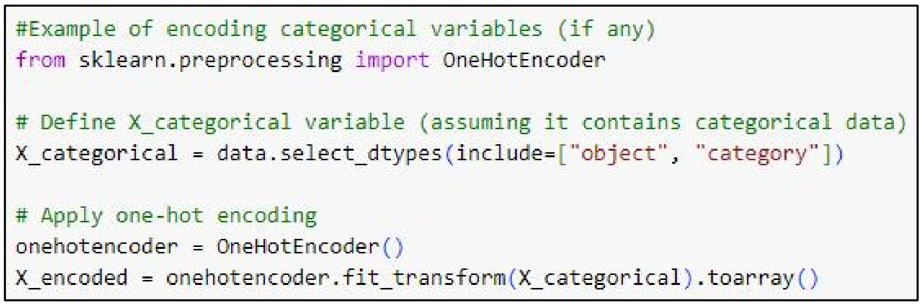


```
Explained Variance by Each Principal Component:
 [9.99997497e-01 2.13847548e-06 3.41602671e-07 1.36012114e-08
 3.96803032e-09 2.22364761e-09 1.44997829e-09 5.52770069e-10
 4.14427159e-10 1.62857193e-10]
```

	0	1	2	3	4	5	6	7	8	9
0	-434908.478025	978.875088	956.538712	28.161058	-10.591170	5.011323	-2.531322	-1.900036	7.169729	-3.530439
1	-434876.496792	928.645471	-248.554484	-63.168067	11.559369	19.676970	-32.819308	43.109329	20.912565	14.468219
2	-434899.831588	928.809832	-248.405864	-62.144303	10.681280	19.304828	-8.812219	27.364308	11.048397	1.640846
3	-434875.082577	928.654000	-248.505183	-62.877819	11.343041	18.369671	-31.505041	40.381094	19.046135	9.020082
4	-434898.417151	928.724914	-247.987972	-59.648116	9.318491	10.998533	-8.171222	-1.553070	-5.308566	-13.731869

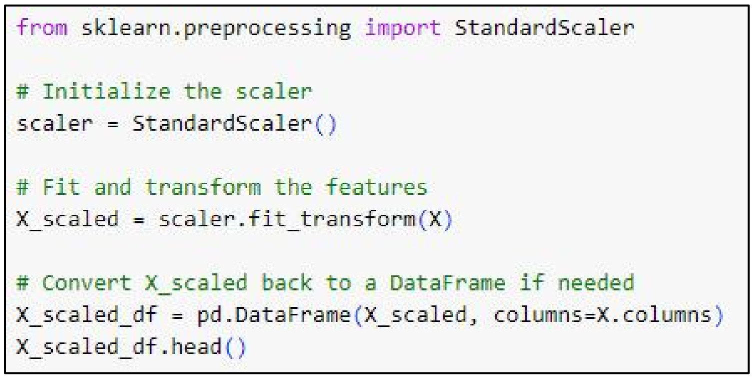
- **Prepare the dataset for model training by encoding categorical variables and scaling numerical features.**

Encode Categorical Variables:
For this dataset, if there are any categorical variables, they need to be encoded. Since all columns are numerical, this step can be skipped. If there are any categorical variables in a different dataset, use one-hot encoding or label encoding.

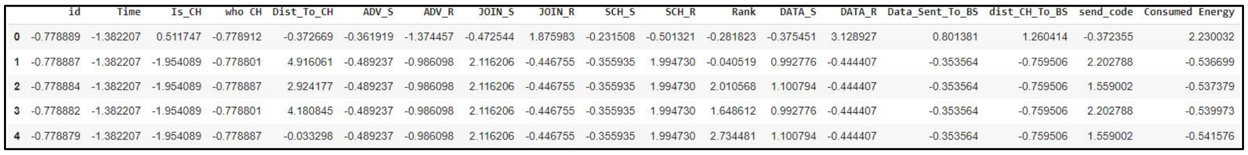


```
#Example of encoding categorical variables (if any)
from sklearn.preprocessing import OneHotEncoder

# Define X_categorical variable (assuming it contains categorical data)
X_categorical = data.select_dtypes(include=["object", "category"])

# Apply one-hot encoding
onehotencoder = OneHotEncoder()
X_encoded = onehotencoder.fit_transform(X_categorical).toarray()
```

Scale Numerical Features:
Scaling numerical features is crucial for many machine learning algorithms. StandardScaler or MinMaxScaler can be used to scale the features.



```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit and transform the features
X_scaled = scaler.fit_transform(X)

# Convert X_scaled back to a DataFrame if needed
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
X_scaled_df.head()
```



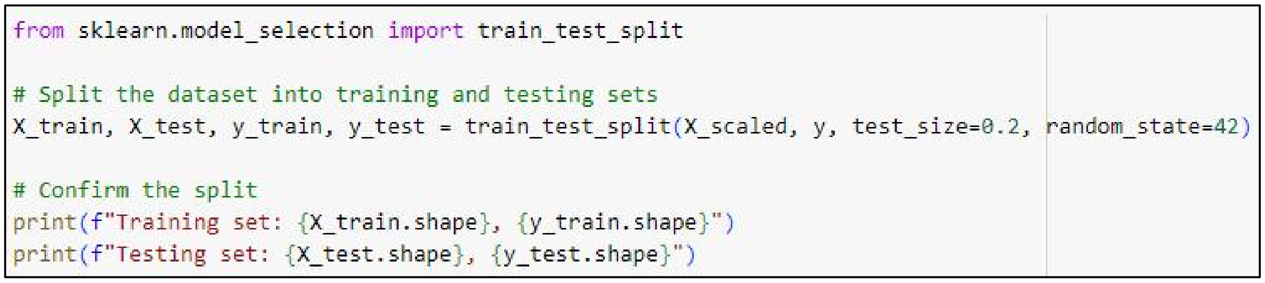
	id	Time	Is_CH	who CH	Dist_To_CH	ADV_S	ADV_R	JOIN_S	JOIN_R	SCH_S	SCH_R	Rank	DATA_S	DATA_R	Data_Sent_To_BS	dist_CH_To_BS	send_code	Consumed Energy
0	-0.778889	-1.382207	0.511747	-0.778912	-0.372669	-0.361919	-1.374457	-0.472544	1.875983	-0.231508	-0.501321	-0.281823	-0.375451	3.128927	0.801381	1.260414	-0.372355	2.230032
1	-0.778887	-1.382207	-1.954089	-0.778801	4.916061	-0.489237	-0.986098	2.116206	-0.446755	-0.355935	1.994730	-0.040519	0.992776	-0.444407	-0.353564	-0.759506	2.202788	-0.536699
2	-0.778884	-1.382207	-1.954089	-0.778887	2.924177	-0.489237	-0.986098	2.116206	-0.446755	-0.355935	1.994730	2.010568	1.100794	-0.444407	-0.353564	-0.759506	1.559002	-0.537379
3	-0.778882	-1.382207	-1.954089	-0.778801	4.180845	-0.489237	-0.986098	2.116206	-0.446755	-0.355935	1.994730	1.648612	0.992776	-0.444407	-0.353564	-0.759506	2.202788	-0.539973
4	-0.778879	-1.382207	-1.954089	-0.778887	-0.033298	-0.489237	-0.986098	2.116206	-0.446755	-0.355935	1.994730	2.734481	1.100794	-0.444407	-0.353564	-0.759506	1.559002	-0.541576

These steps prepare the dataset for model training, ensuring that the features are in a suitable format and scale for machine learning algorithms. Next, we can proceed to model selection, training, and evaluation.

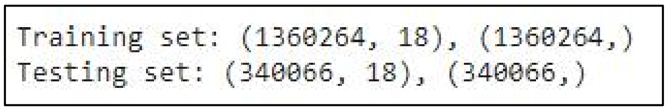
Step 4: Model Selection and Building

- **Split the preprocessed dataset into training and testing sets to enable model evaluation.**

First, split the preprocessed dataset into training and testing sets to enable model evaluation.



```
from sklearn.model_selection import train_test_split

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Confirm the split
print(f"Training set: {X_train.shape}, {y_train.shape}")
print(f"Testing set: {X_test.shape}, {y_test.shape}")
```



```
Training set: (1360264, 18), (1360264,)
Testing set: (340066, 18), (340066,)
```

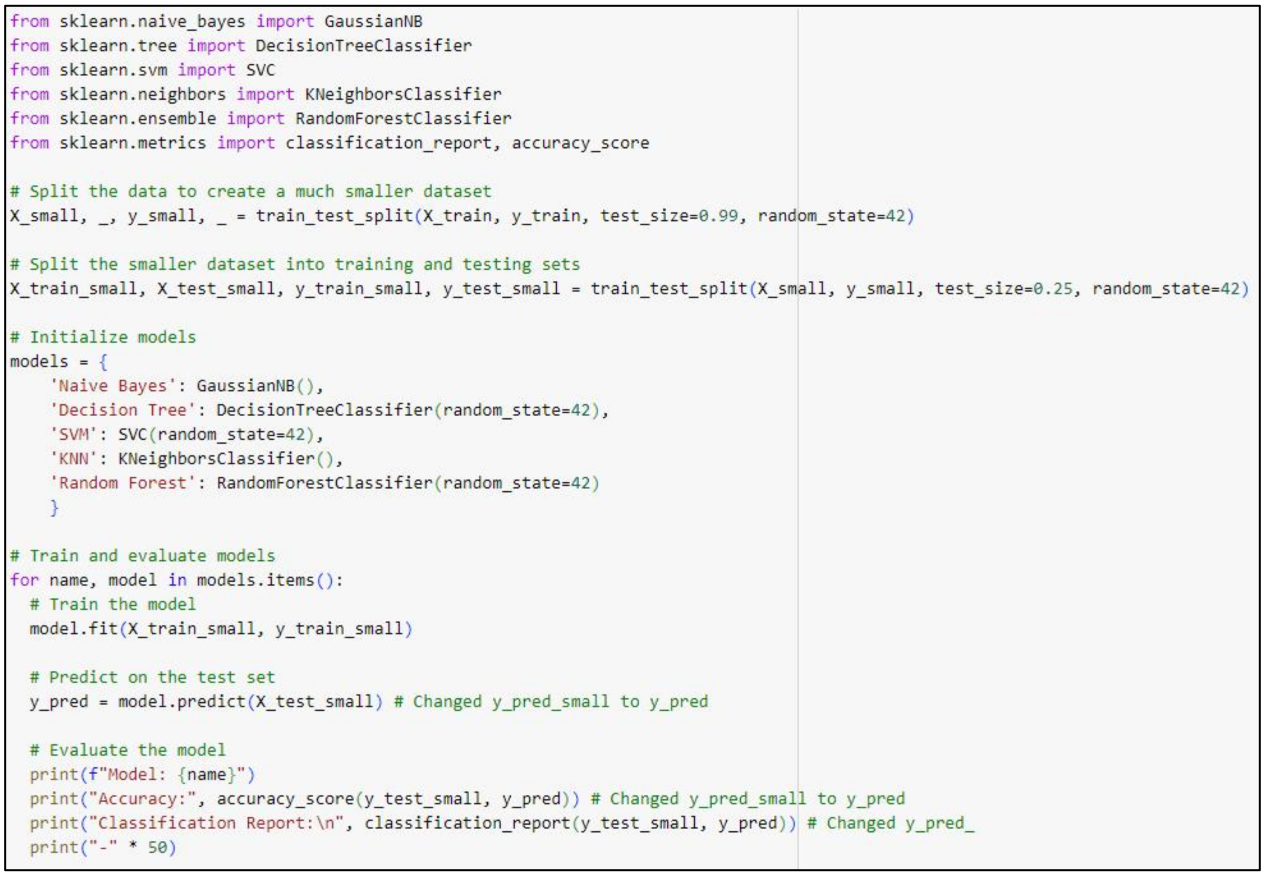
- **Choose at least five classification algorithms (e.g., Naïve Bayes, Decision Trees, Support Vector Machine, KNN, and Random Forest) for comparison.**

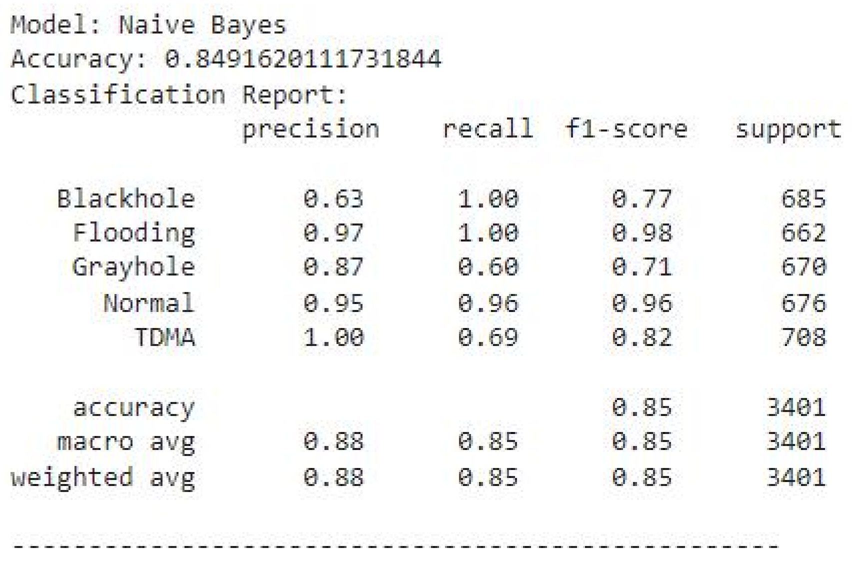
We will use the following five classification algorithms for comparison:

1. Naïve Bayes
2. Decision Trees
3. Support Vector Machine (SVM)
4. K-Nearest Neighbors (KNN)
5. Random Forest

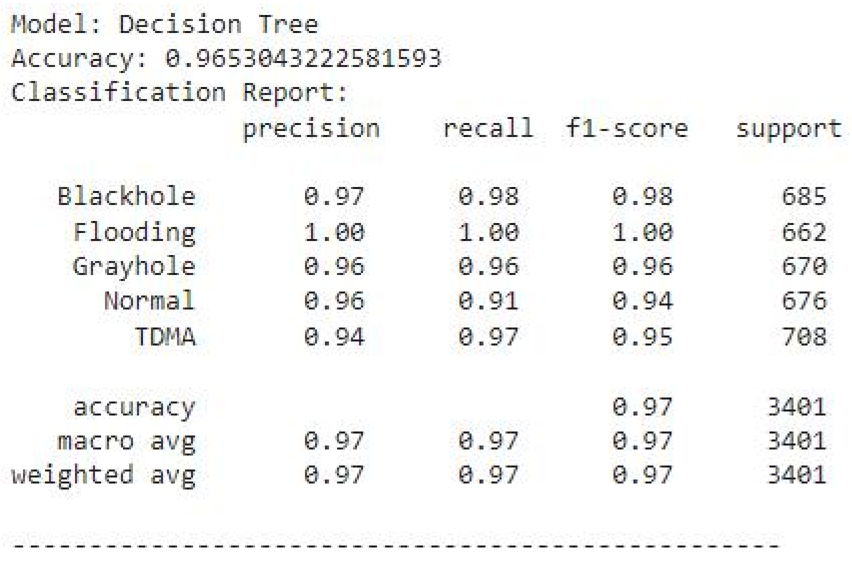
- **Implement each chosen algorithm using appropriate libraries in Python, and train the models using the training dataset.**

Implement each algorithm using appropriate libraries in Python, and train the models using the training dataset.

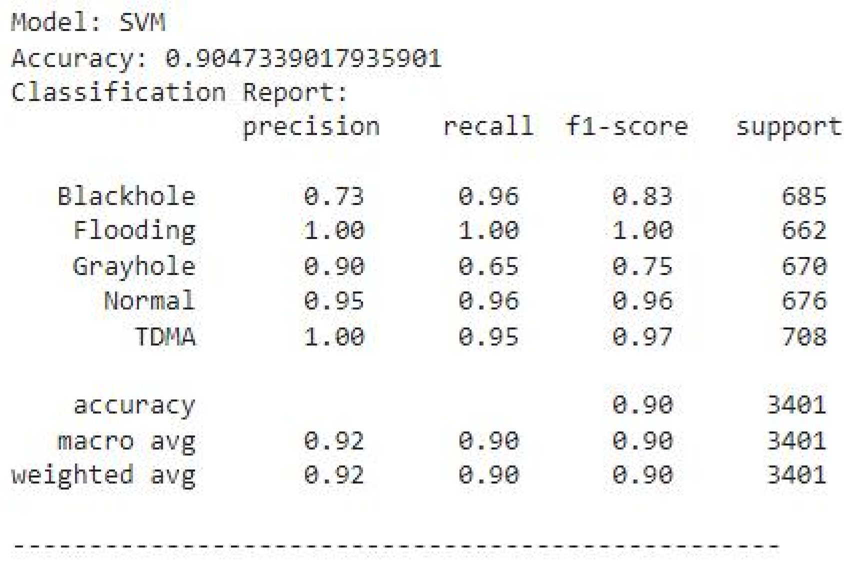

```
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score

# Split the data to create a much smaller dataset
X_small, _, y_small, _ = train_test_split(X_train, y_train, test_size=0.99, random_state=42)

# Split the smaller dataset into training and testing sets
X_train_small, X_test_small, y_train_small, y_test_small = train_test_split(X_small, y_small, test_size=0.25, random_state=42)

# Initialize models
models = {
    'Naive Bayes': GaussianNB(),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'SVM': SVC(random_state=42),
    'KNN': KNeighborsClassifier(),
    'Random Forest': RandomForestClassifier(random_state=42)
    }

# Train and evaluate models
for name, model in models.items():
  # Train the model
  model.fit(X_train_small, y_train_small)

  # Predict on the test set
  y_pred = model.predict(X_test_small) # Changed y_pred_small to y_pred

  # Evaluate the model
  print(f"Model: {name}")
  print("Accuracy:", accuracy_score(y_test_small, y_pred)) # Changed y_pred_small to y_pred
  print("Classification Report:\n", classification_report(y_test_small, y_pred)) # Changed y_pred_
  print("-" * 50)
```



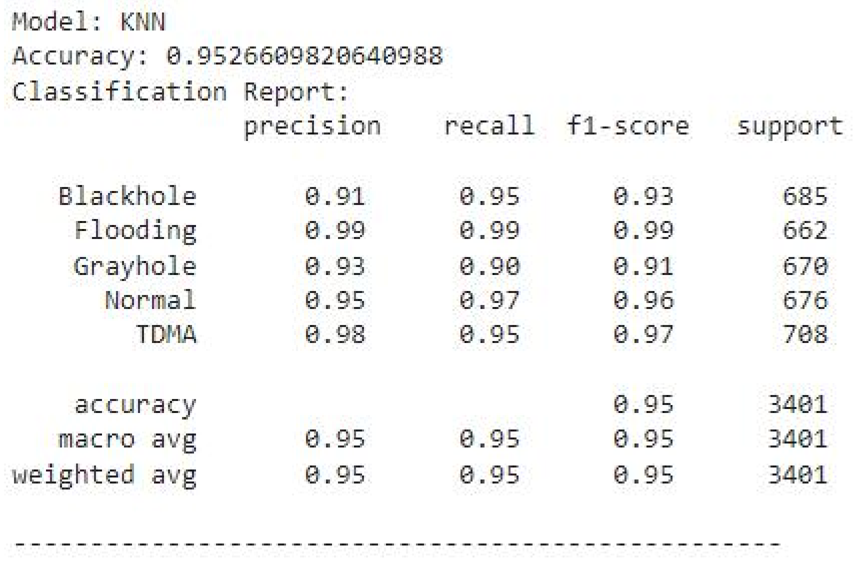
```
Model: Naive Bayes
Accuracy: 0.8491620111731844
Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.63      1.00      0.77       685
    Flooding       0.97      1.00      0.98       662
    Grayhole       0.87      0.60      0.71       670
      Normal       0.95      0.96      0.96       676
        TDMA       1.00      0.69      0.82       708

    accuracy                           0.85      3401
   macro avg       0.88      0.85      0.85      3401
weighted avg       0.88      0.85      0.85      3401

--------------------------------------------------
```



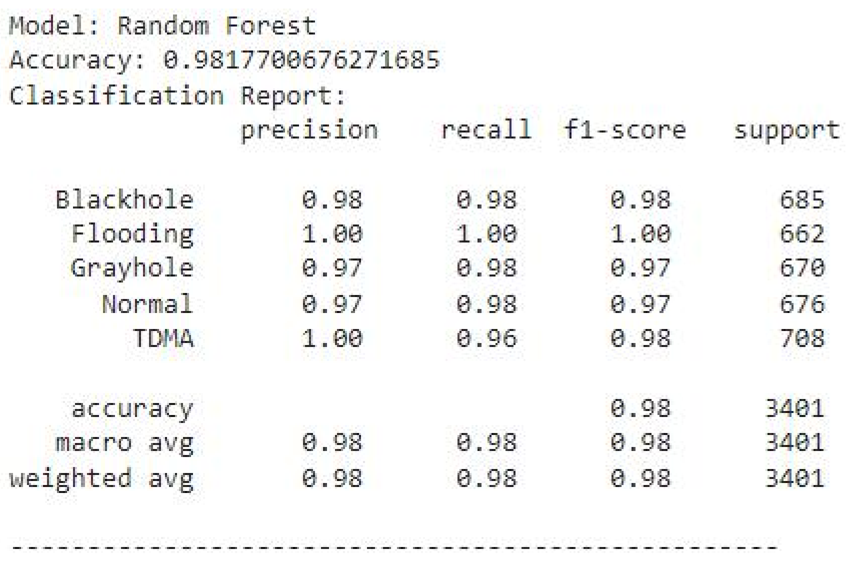
```
Model: Decision Tree
Accuracy: 0.9653043222581593
Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.97      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.91      0.94       676
        TDMA       0.94      0.97      0.95       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
```



```
Model: SVM
Accuracy: 0.9047339017935901
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.73      0.96      0.83       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.90      0.65      0.75       670
      Normal       0.95      0.96      0.96       676
        TDMA       1.00      0.95      0.97       708

    accuracy                           0.90      3401
   macro avg       0.92      0.90      0.90      3401
weighted avg       0.92      0.90      0.90      3401

--------------------------------------------------
```



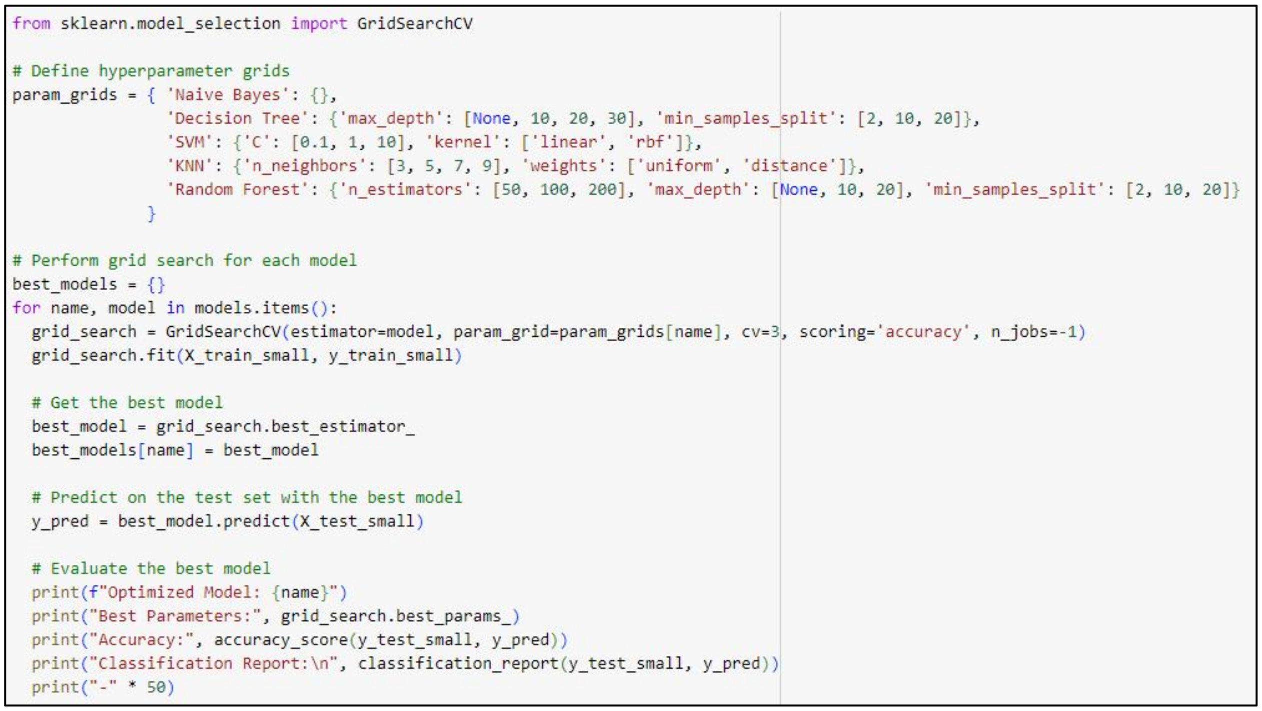
```
Model: KNN
Accuracy: 0.9526609820640988
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.91      0.95      0.93       685
    Flooding       0.99      0.99      0.99       662
    Grayhole       0.93      0.90      0.91       670
      Normal       0.95      0.97      0.96       676
        TDMA       0.98      0.95      0.97       708

    accuracy                           0.95      3401
   macro avg       0.95      0.95      0.95      3401
weighted avg       0.95      0.95      0.95      3401

--------------------------------------------------
```



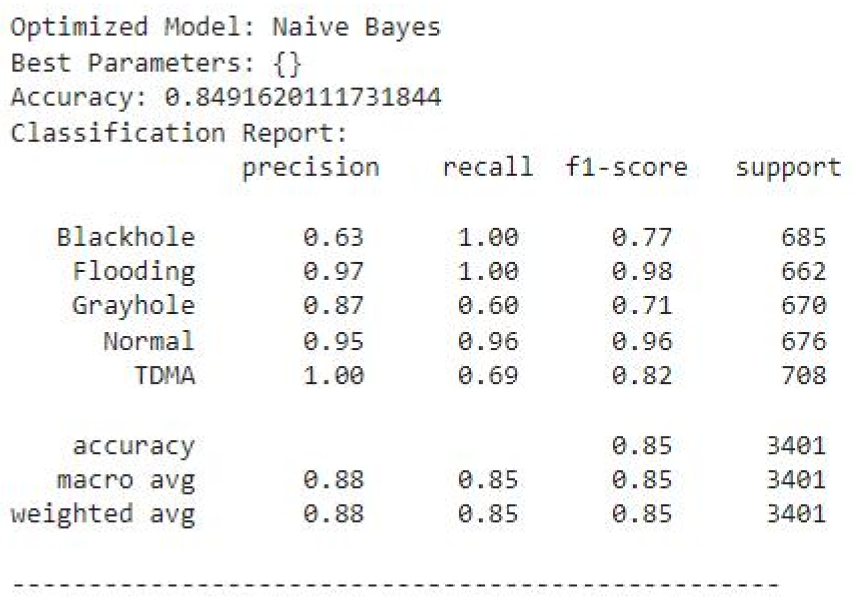
```
Model: Random Forest
Accuracy: 0.9817700676271685
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.97       670
      Normal       0.97      0.98      0.97       676
        TDMA       1.00      0.96      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```

- **Optimize hyper parameters for each model to improve classification performance, using techniques such as grid search or random search.**

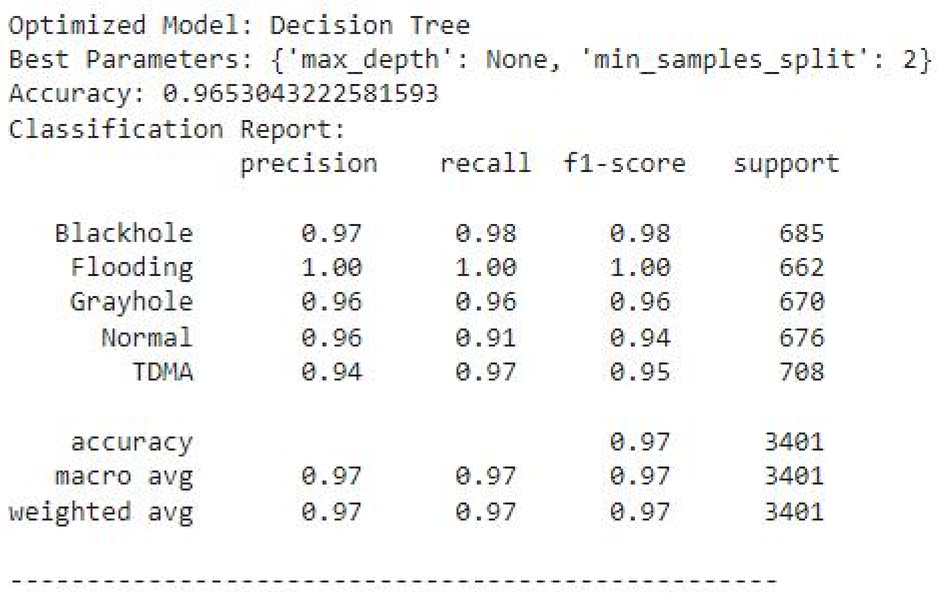
Use techniques such as grid search or random search to optimize hyper parameters for each model. Here, we'll use `GridSearchCV` for hyper parameter tuning.



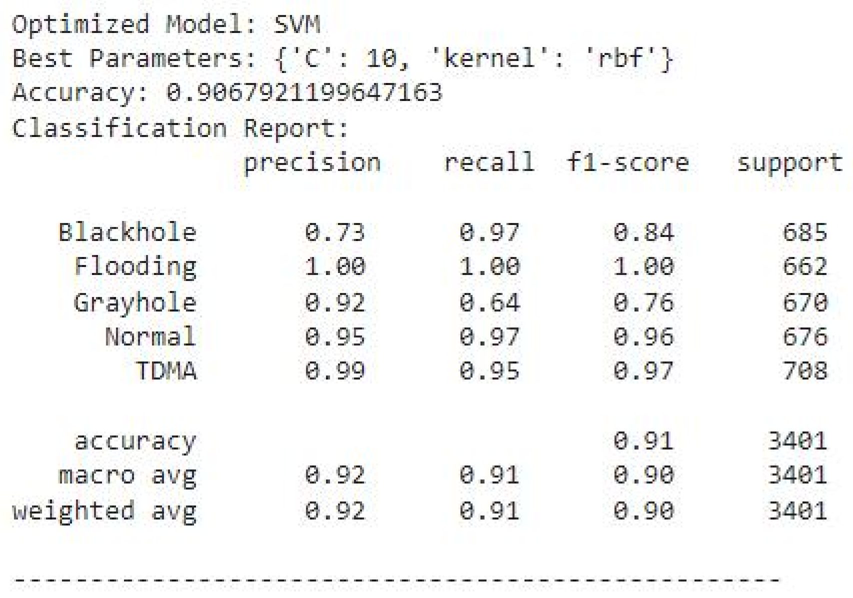
```python
from sklearn.model_selection import GridSearchCV

# Define hyperparameter grids
param_grids = { 'Naive Bayes': {},
                'Decision Tree': {'max_depth': [None, 10, 20, 30], 'min_samples_split': [2, 10, 20]},
                'SVM': {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']},
                'KNN': {'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform', 'distance']},
                'Random Forest': {'n_estimators': [50, 100, 200], 'max_depth': [None, 10, 20], 'min_samples_split': [2, 10, 20]}
              }

# Perform grid search for each model
best_models = {}
for name, model in models.items():
  grid_search = GridSearchCV(estimator=model, param_grid=param_grids[name], cv=3, scoring='accuracy', n_jobs=-1)
  grid_search.fit(X_train_small, y_train_small)

  # Get the best model
  best_model = grid_search.best_estimator_
  best_models[name] = best_model

  # Predict on the test set with the best model
  y_pred = best_model.predict(X_test_small)

  # Evaluate the best model
  print(f"Optimized Model: {name}")
  print("Best Parameters:", grid_search.best_params_)
  print("Accuracy:", accuracy_score(y_test_small, y_pred))
  print("Classification Report:\n", classification_report(y_test_small, y_pred))
  print("-" * 50)
```



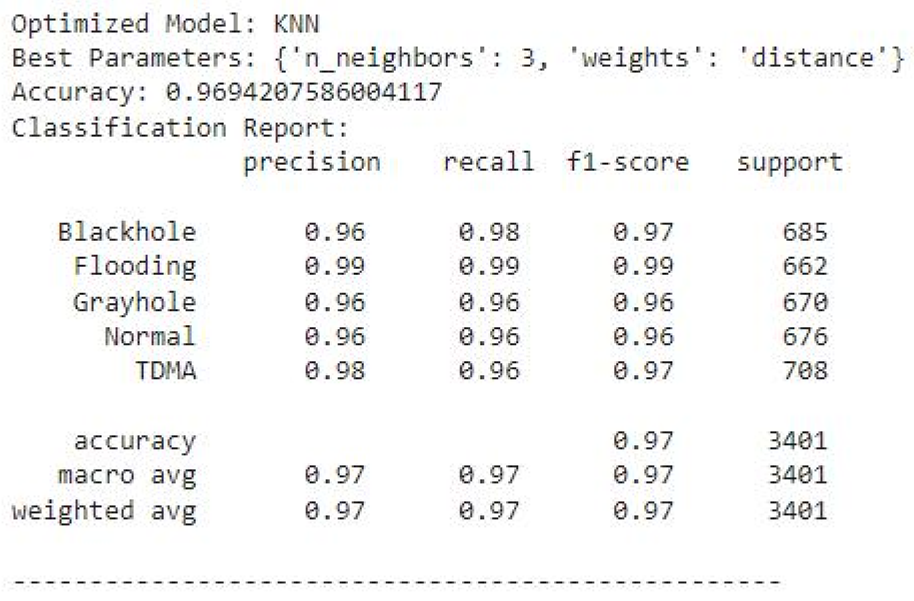
```
Optimized Model: Naive Bayes
Best Parameters: {}
Accuracy: 0.8491620111731844
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.63      1.00      0.77       685
    Flooding       0.97      1.00      0.98       662
    Grayhole       0.87      0.60      0.71       670
      Normal       0.95      0.96      0.96       676
        TDMA       1.00      0.69      0.82       708

    accuracy                           0.85      3401
   macro avg       0.88      0.85      0.85      3401
weighted avg       0.88      0.85      0.85      3401

--------------------------------------------------
```



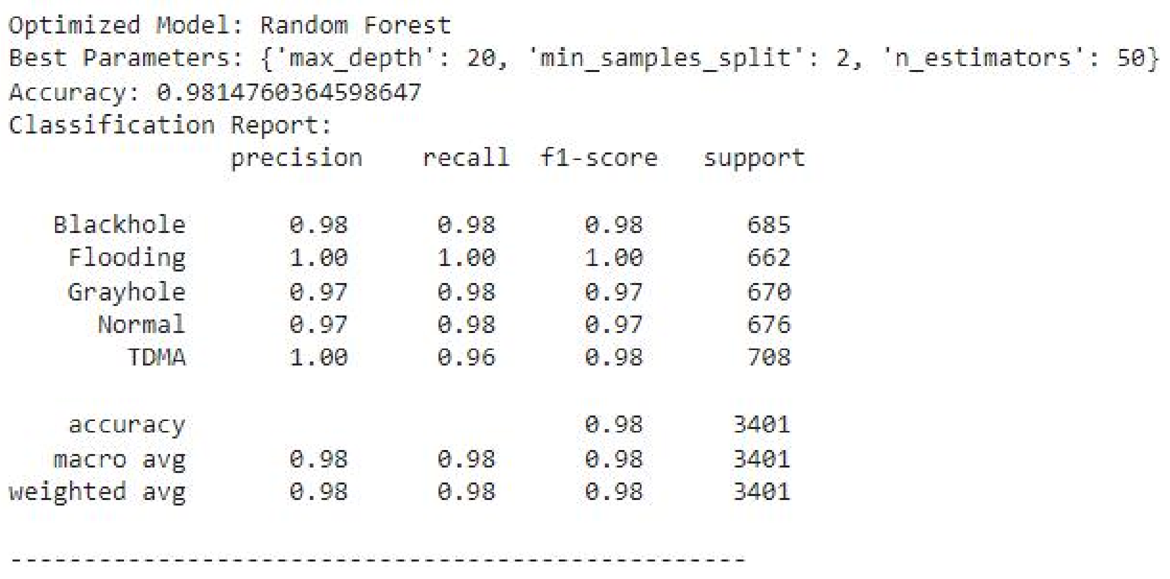
```
Optimized Model: Decision Tree
Best Parameters: {'max_depth': None, 'min_samples_split': 2}
Accuracy: 0.9653043222581593
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.97      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.91      0.94       676
        TDMA       0.94      0.97      0.95       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
```



```
Optimized Model: SVM
Best Parameters: {'C': 10, 'kernel': 'rbf'}
Accuracy: 0.9067921199647163
Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.73      0.97      0.84       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.92      0.64      0.76       670
      Normal       0.95      0.97      0.96       676
        TDMA       0.99      0.95      0.97       708

    accuracy                           0.91      3401
   macro avg       0.92      0.91      0.90      3401
weighted avg       0.92      0.91      0.90      3401

--------------------------------------------------
```



```
Optimized Model: KNN
Best Parameters: {'n_neighbors': 3, 'weights': 'distance'}
Accuracy: 0.9694207586004117
Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.96      0.98      0.97       685
    Flooding       0.99      0.99      0.99       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.96      0.96       676
        TDMA       0.98      0.96      0.97       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
```



```
Optimized Model: Random Forest
Best Parameters: {'max_depth': 20, 'min_samples_split': 2, 'n_estimators': 50}
Accuracy: 0.9814760364598647
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.97       670
      Normal       0.97      0.98      0.97       676
        TDMA       1.00      0.96      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```

These steps help in building and selecting the best model for classifying DoS attacks in Wireless Sensor Networks (WSN). Next, you can proceed to the final step of model evaluation and selection.

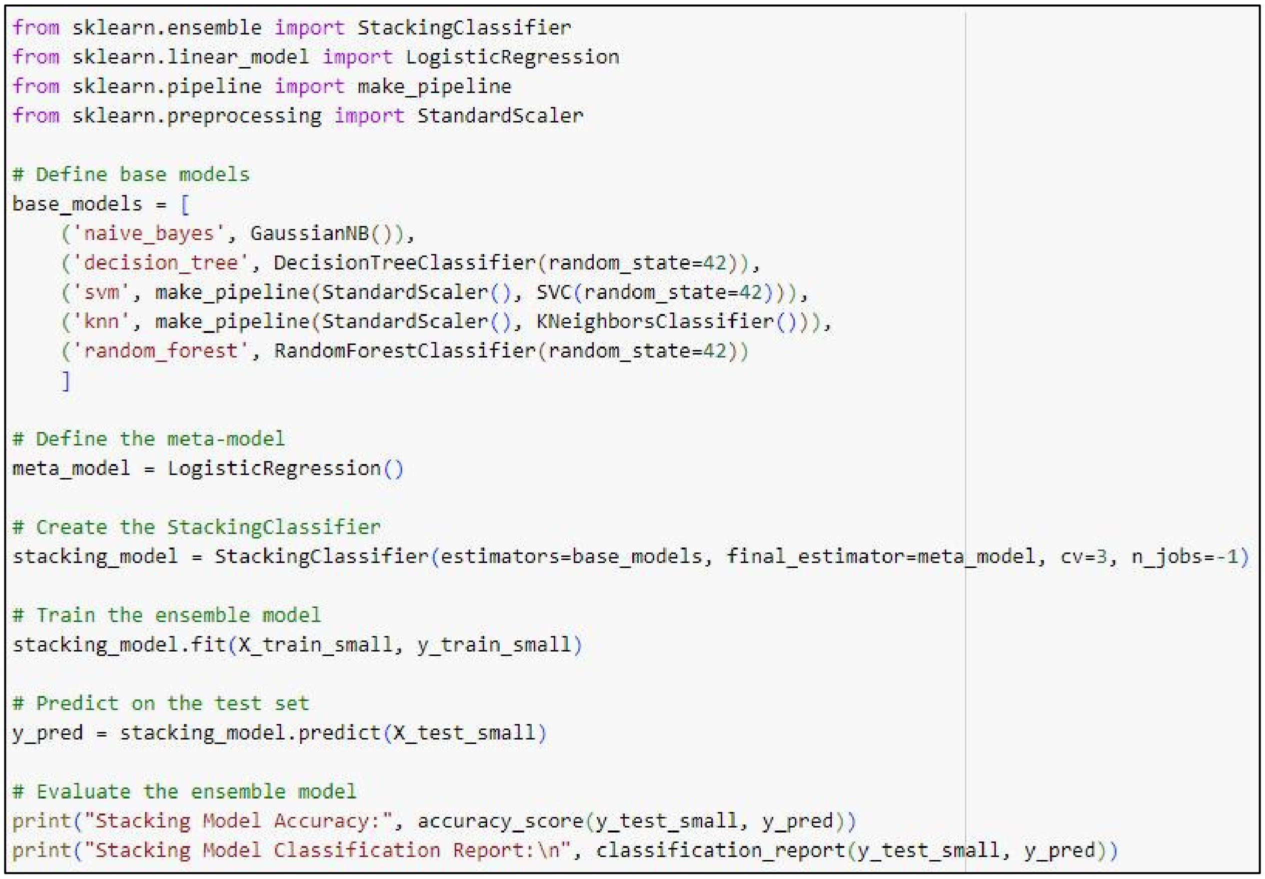
Step 5: Ensemble Model Building

- **Choose a suitable ensemble method (Bagging, Boosting, or Stacking).**

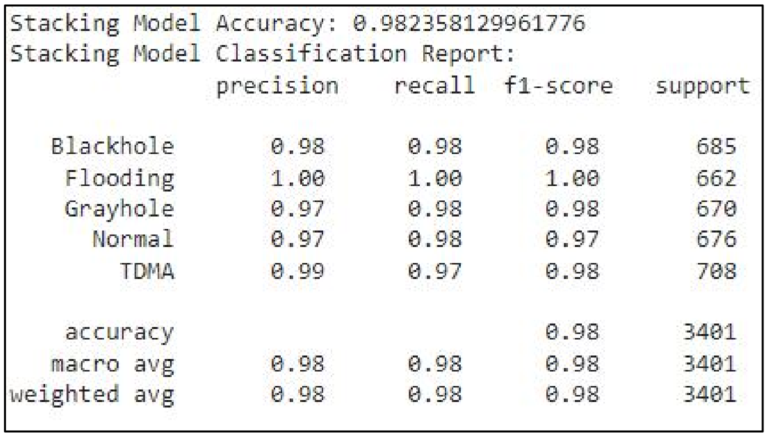
For this step, we'll choose **Stacking** as our ensemble method. Stacking involves training multiple base models and then using a meta-model to combine their predictions.

- **Implement the ensemble model using appropriate libraries in Python, and train it using the training dataset.**

We will use the `StackingClassifier` from `sklearn` to build our ensemble model.



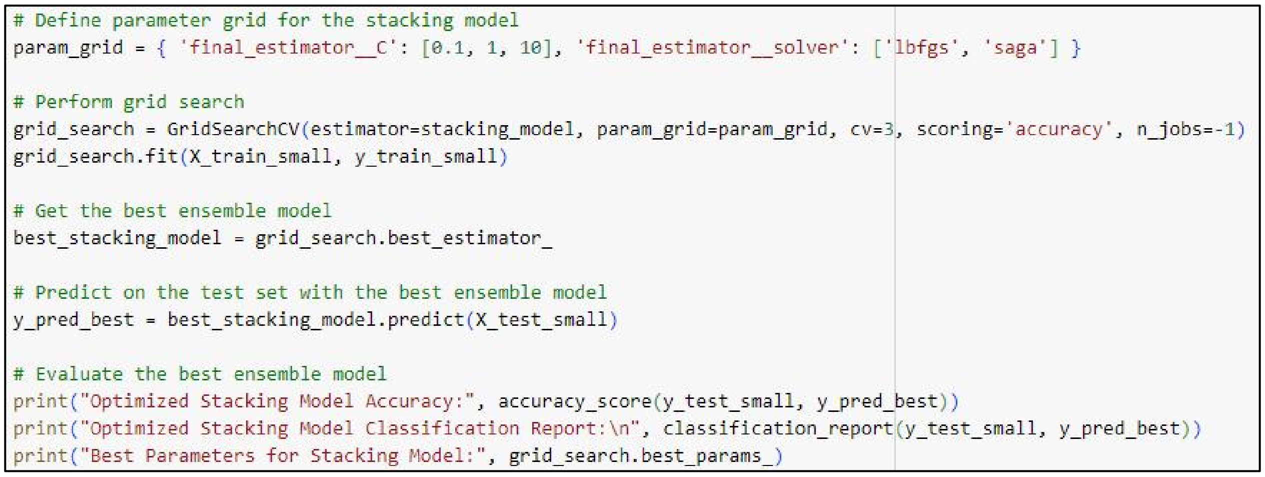
```python
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# Define base models
base_models = [
    ('naive_bayes', GaussianNB()),
    ('decision_tree', DecisionTreeClassifier(random_state=42)),
    ('svm', make_pipeline(StandardScaler(), SVC(random_state=42))),
    ('knn', make_pipeline(StandardScaler(), KNeighborsClassifier())),
    ('random_forest', RandomForestClassifier(random_state=42))
    ]

# Define the meta-model
meta_model = LogisticRegression()

# Create the StackingClassifier
stacking_model = StackingClassifier(estimators=base_models, final_estimator=meta_model, cv=3, n_jobs=-1)

# Train the ensemble model
stacking_model.fit(X_train_small, y_train_small)

# Predict on the test set
y_pred = stacking_model.predict(X_test_small)

# Evaluate the ensemble model
print("Stacking Model Accuracy:", accuracy_score(y_test_small, y_pred))
print("Stacking Model Classification Report:\n", classification_report(y_test_small, y_pred))
```



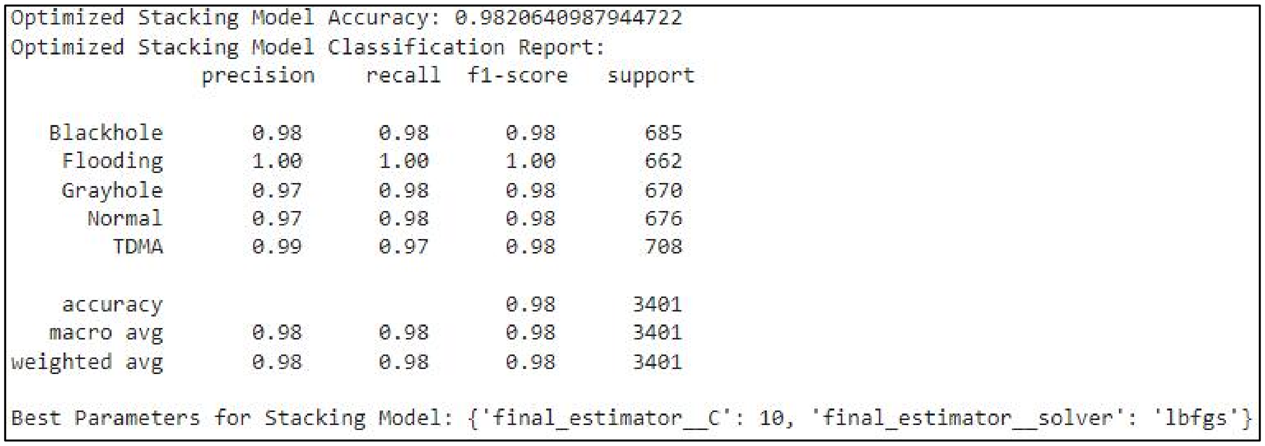
```
Stacking Model Accuracy: 0.982358129961776
Stacking Model Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.97       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401
```

- **Tune the hyper parameters of the ensemble model to optimize its performance.**

Use grid search to tune the hyper parameters of the ensemble model for optimal performance.



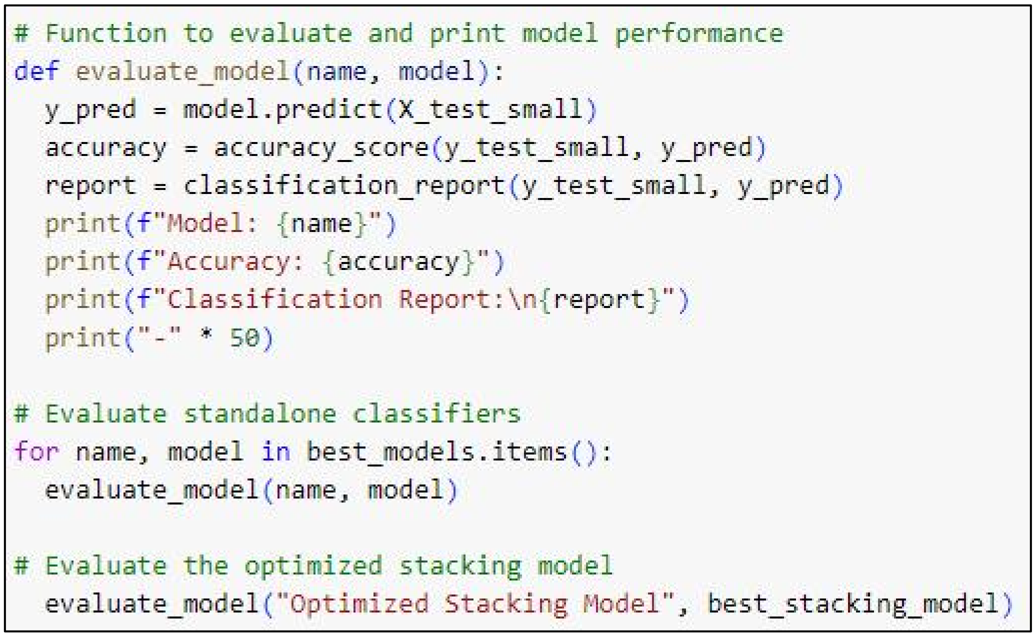
```
# Define parameter grid for the stacking model
param_grid = { 'final_estimator__C': [0.1, 1, 10], 'final_estimator__solver': ['lbfgs', 'saga'] }

# Perform grid search
grid_search = GridSearchCV(estimator=stacking_model, param_grid=param_grid, cv=3, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train_small, y_train_small)

# Get the best ensemble model
best_stacking_model = grid_search.best_estimator_

# Predict on the test set with the best ensemble model
y_pred_best = best_stacking_model.predict(X_test_small)

# Evaluate the best ensemble model
print("Optimized Stacking Model Accuracy:", accuracy_score(y_test_small, y_pred_best))
print("Optimized Stacking Model Classification Report:\n", classification_report(y_test_small, y_pred_best))
print("Best Parameters for Stacking Model:", grid_search.best_params_)
```



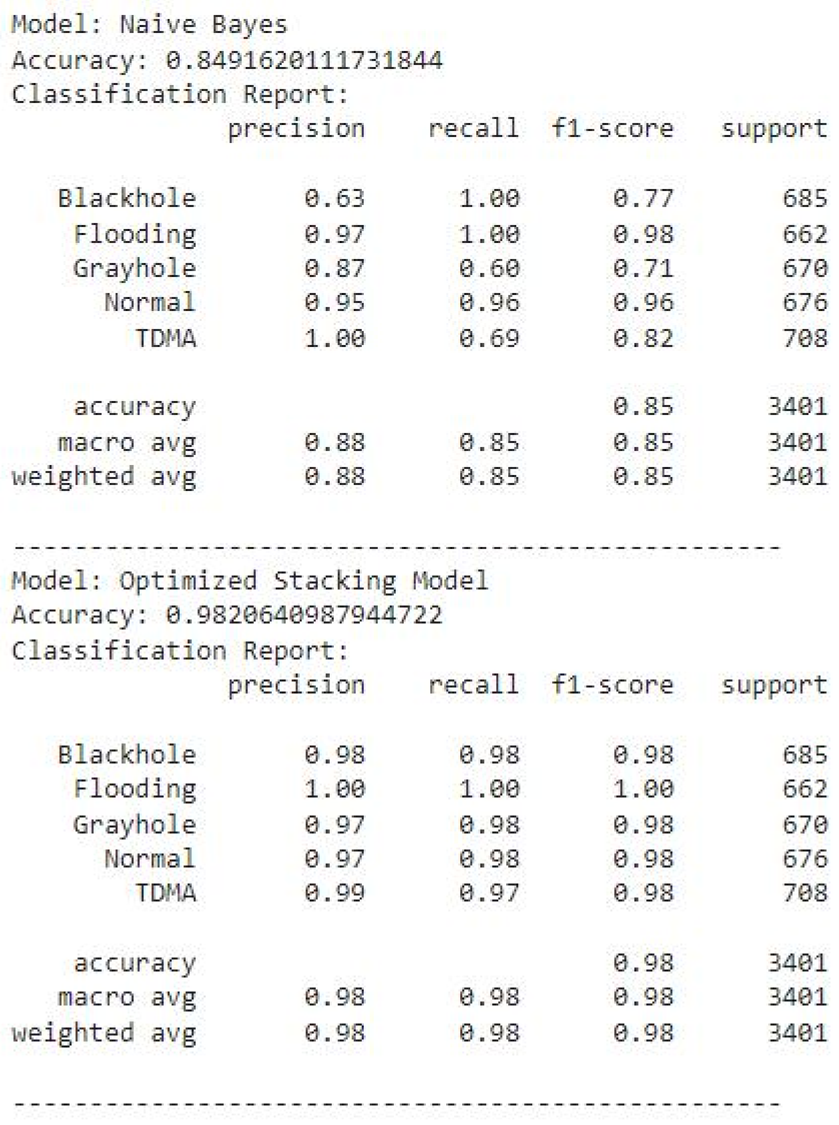
```
Optimized Stacking Model Accuracy: 0.9820640987944722
Optimized Stacking Model Classification Report:
               precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

Best Parameters for Stacking Model: {'final_estimator__C': 10, 'final_estimator__solver': 'lbfgs'}
```

- **Compare the performance of the ensemble model with standalone classifiers (Naïve Bayes, Decision Trees, Support Vector Machine, KNN, Random Forest) using appropriate evaluation metrics.**

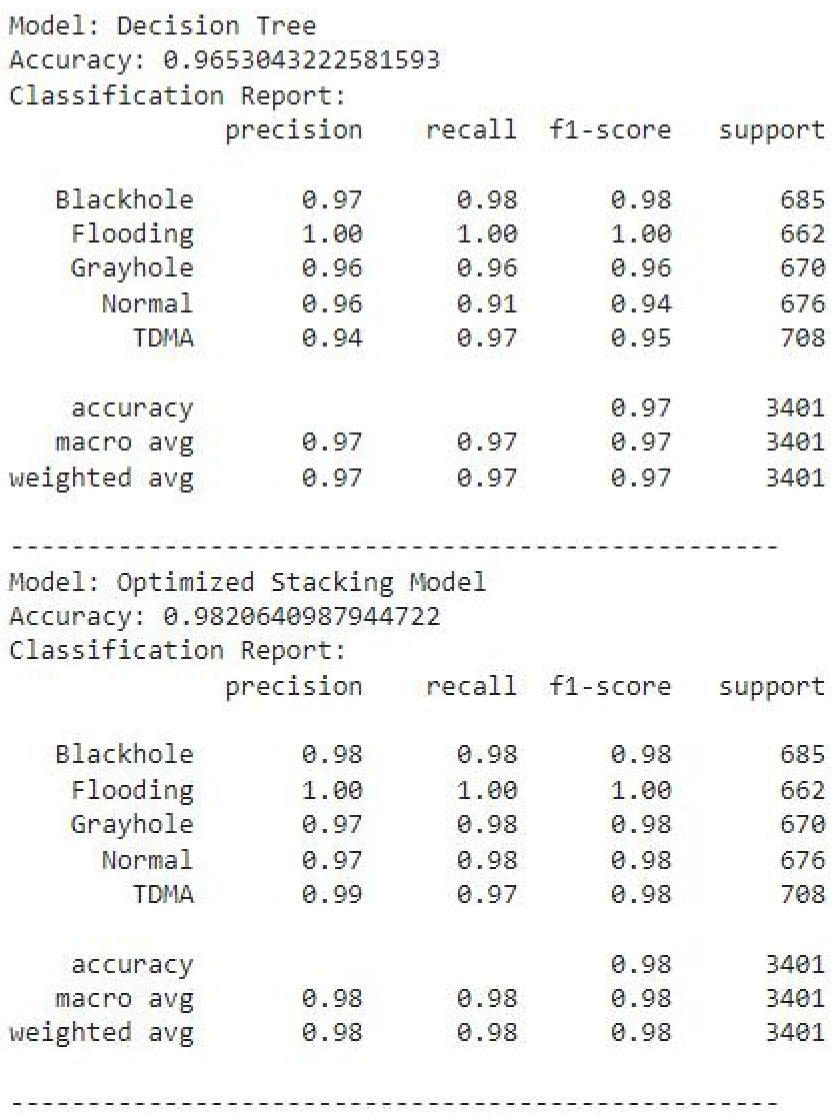
We will compare the performance of the ensemble model with the standalone classifiers using the evaluation metrics we gathered earlier.



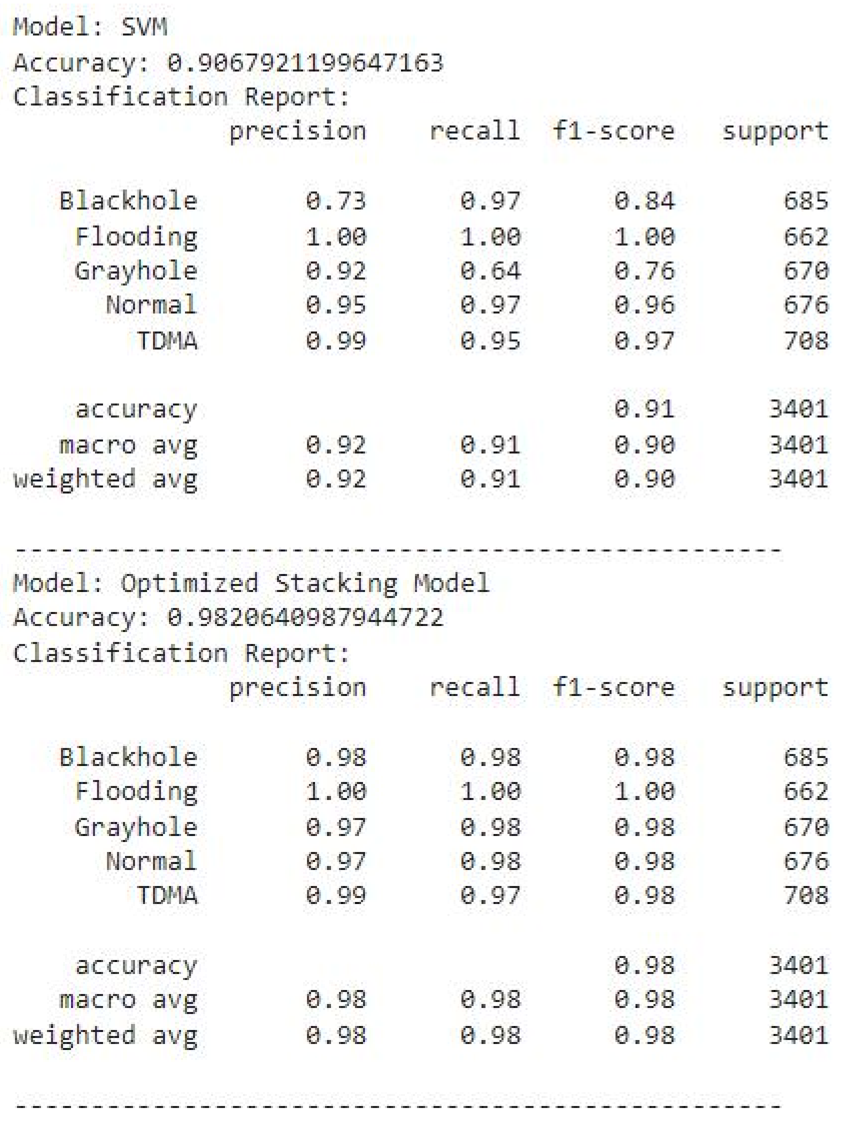
```
# Function to evaluate and print model performance
def evaluate_model(name, model):
  y_pred = model.predict(X_test_small)
  accuracy = accuracy_score(y_test_small, y_pred)
  report = classification_report(y_test_small, y_pred)
  print(f"Model: {name}")
  print(f"Accuracy: {accuracy}")
  print(f"Classification Report:\n{report}")
  print("-" * 50)

# Evaluate standalone classifiers
for name, model in best_models.items():
  evaluate_model(name, model)

# Evaluate the optimized stacking model
  evaluate_model("Optimized Stacking Model", best_stacking_model)
```



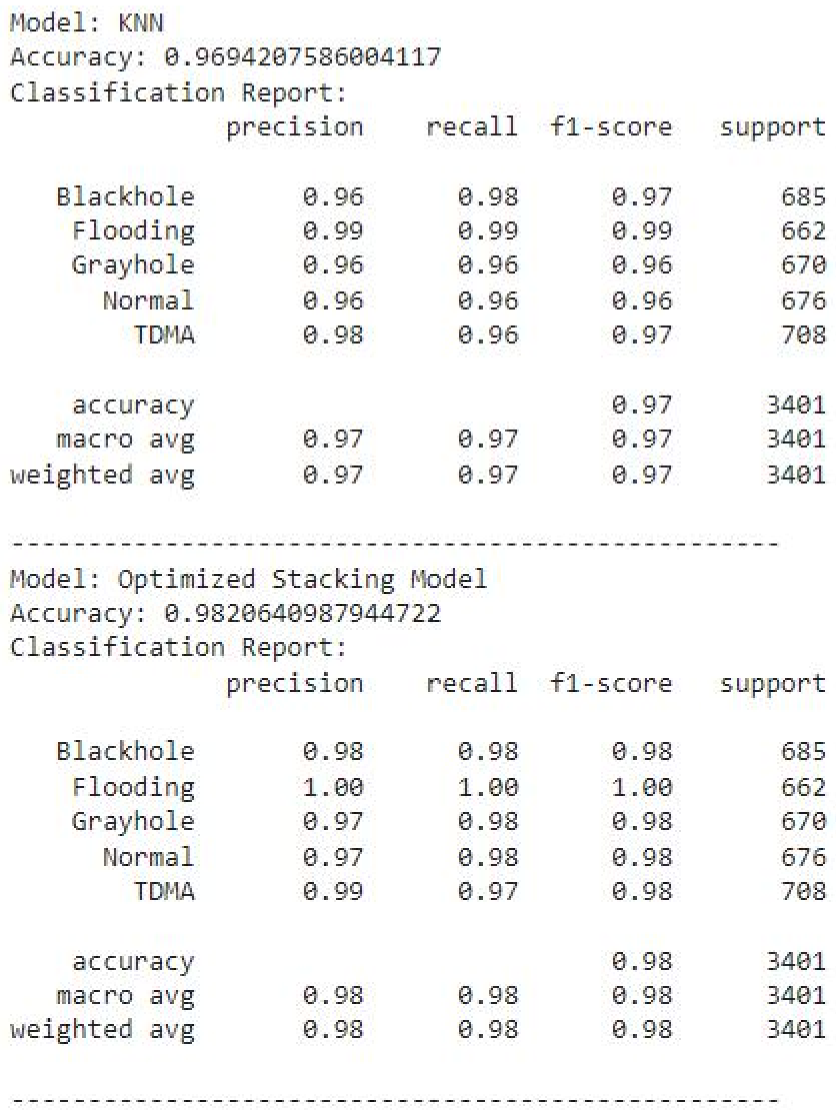
```
Model: Naive Bayes
Accuracy: 0.8491620111731844
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.63      1.00      0.77       685
    Flooding       0.97      1.00      0.98       662
    Grayhole       0.87      0.60      0.71       670
      Normal       0.95      0.96      0.96       676
        TDMA       1.00      0.69      0.82       708

    accuracy                           0.85      3401
   macro avg       0.88      0.85      0.85      3401
weighted avg       0.88      0.85      0.85      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```



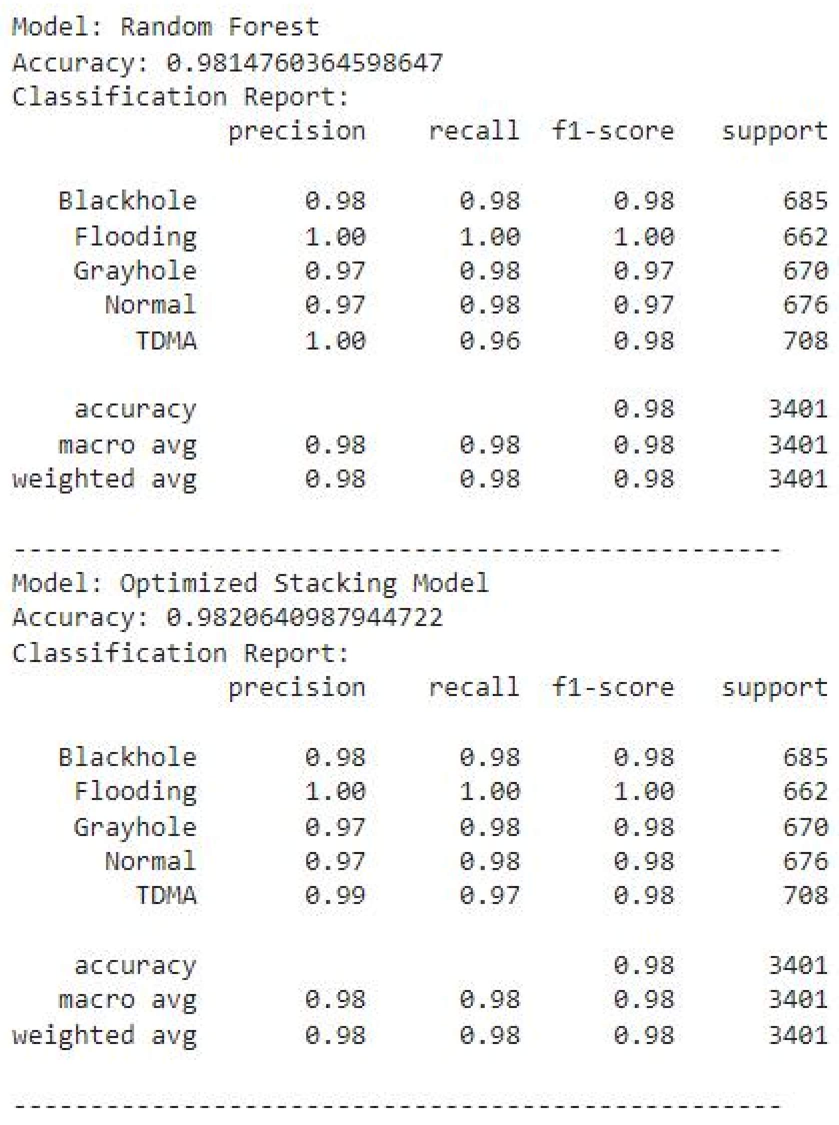
```
Model: Decision Tree
Accuracy: 0.9653043222581593
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.97      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.91      0.94       676
        TDMA       0.94      0.97      0.95       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```



```
Model: SVM
Accuracy: 0.9067921199647163
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.73      0.97      0.84       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.92      0.64      0.76       670
      Normal       0.95      0.97      0.96       676
        TDMA       0.99      0.95      0.97       708

    accuracy                           0.91      3401
   macro avg       0.92      0.91      0.90      3401
weighted avg       0.92      0.91      0.90      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```



```
Model: KNN
Accuracy: 0.9694207586004117
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.96      0.98      0.97       685
    Flooding       0.99      0.99      0.99       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.96      0.96       676
        TDMA       0.98      0.96      0.97       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```



```
Model: Random Forest
Accuracy: 0.9814760364598647
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.97       670
      Normal       0.97      0.98      0.97       676
        TDMA       1.00      0.96      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```

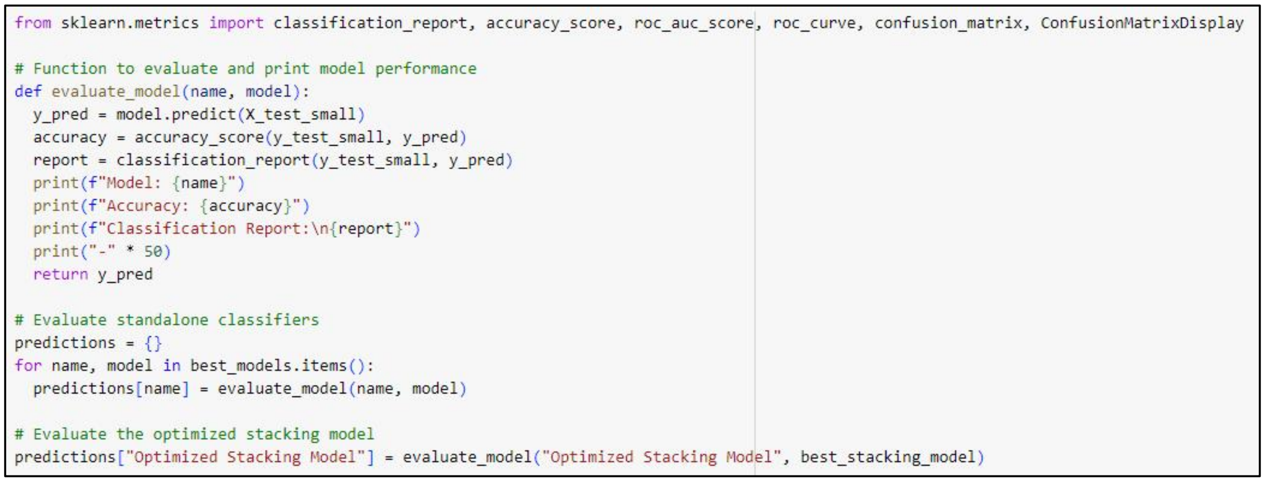
These steps ensure that we have built an optimized ensemble model and compared its performance with individual classifiers to select the best model for classifying DoS attacks in Wireless Sensor Networks.

Step 6: Model Evaluation

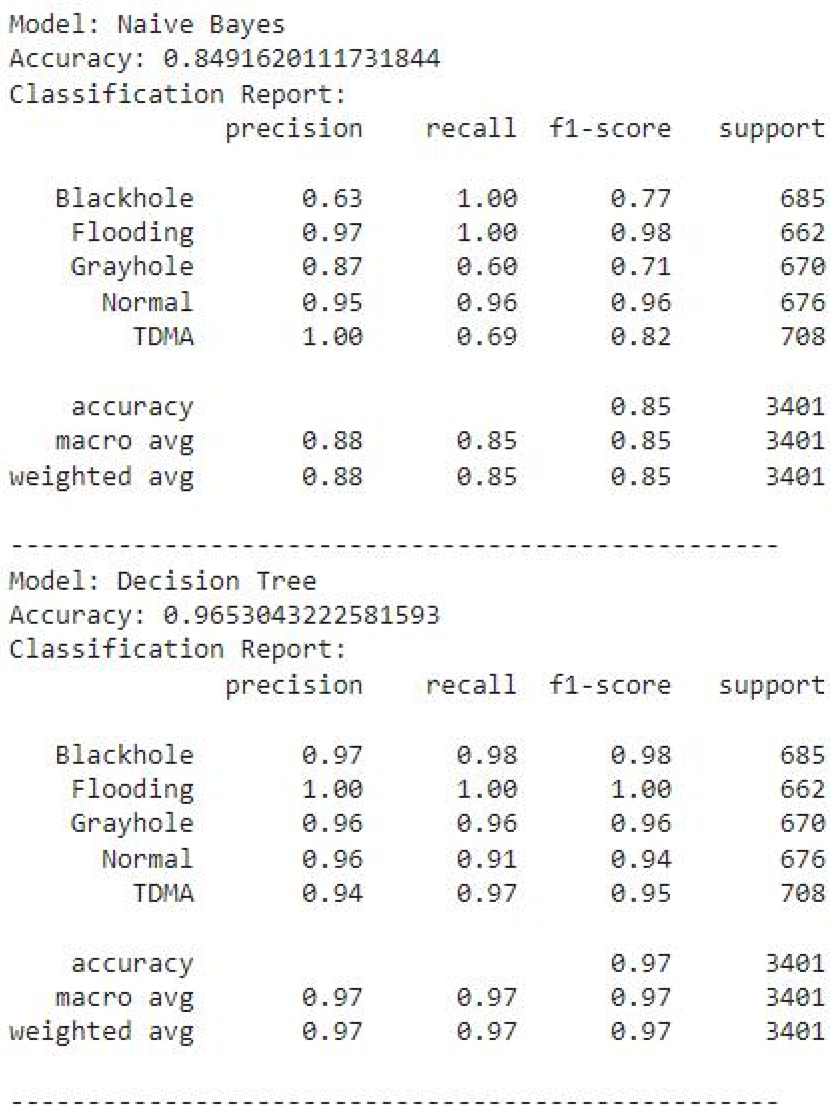
- **Evaluate the performance of each model using the testing dataset.**

We'll evaluate the performance of each model using the testing dataset, generate classification reports, and compute the AUC-ROC curve.

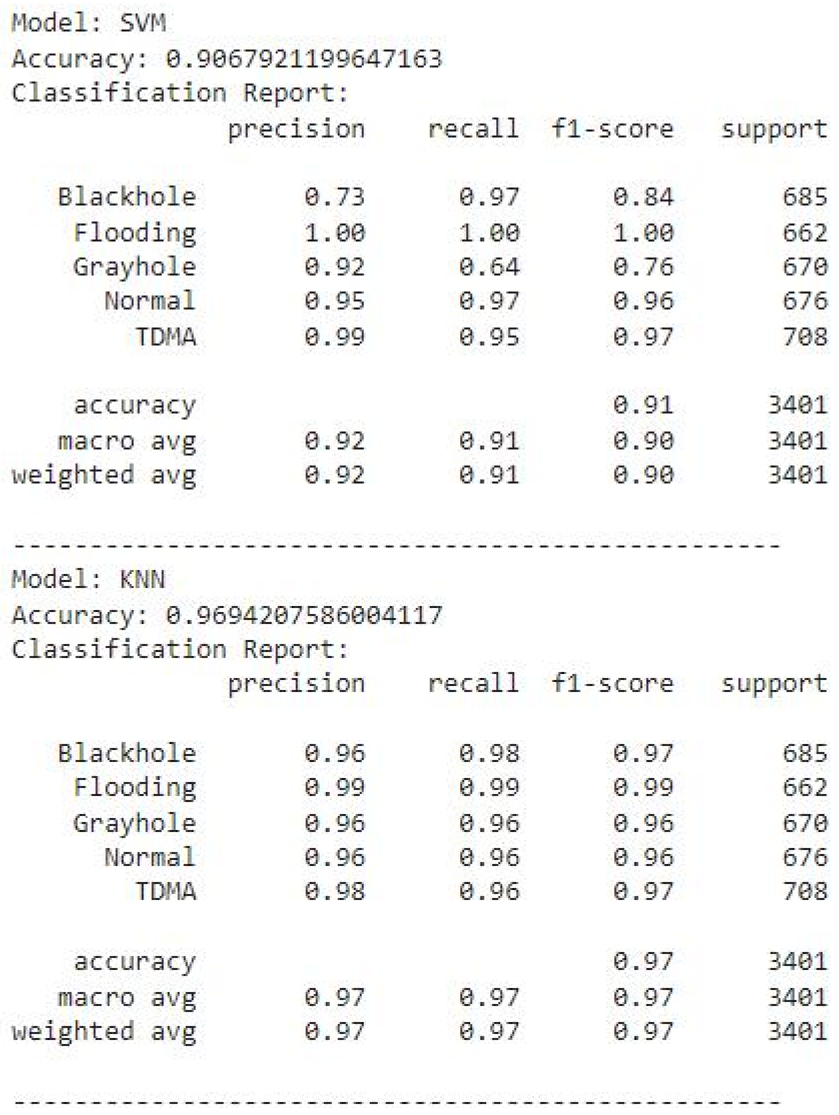
- **Generate a classification report for each model, including metrics such as precision, recall, F1-score, and accuracy for each class.**



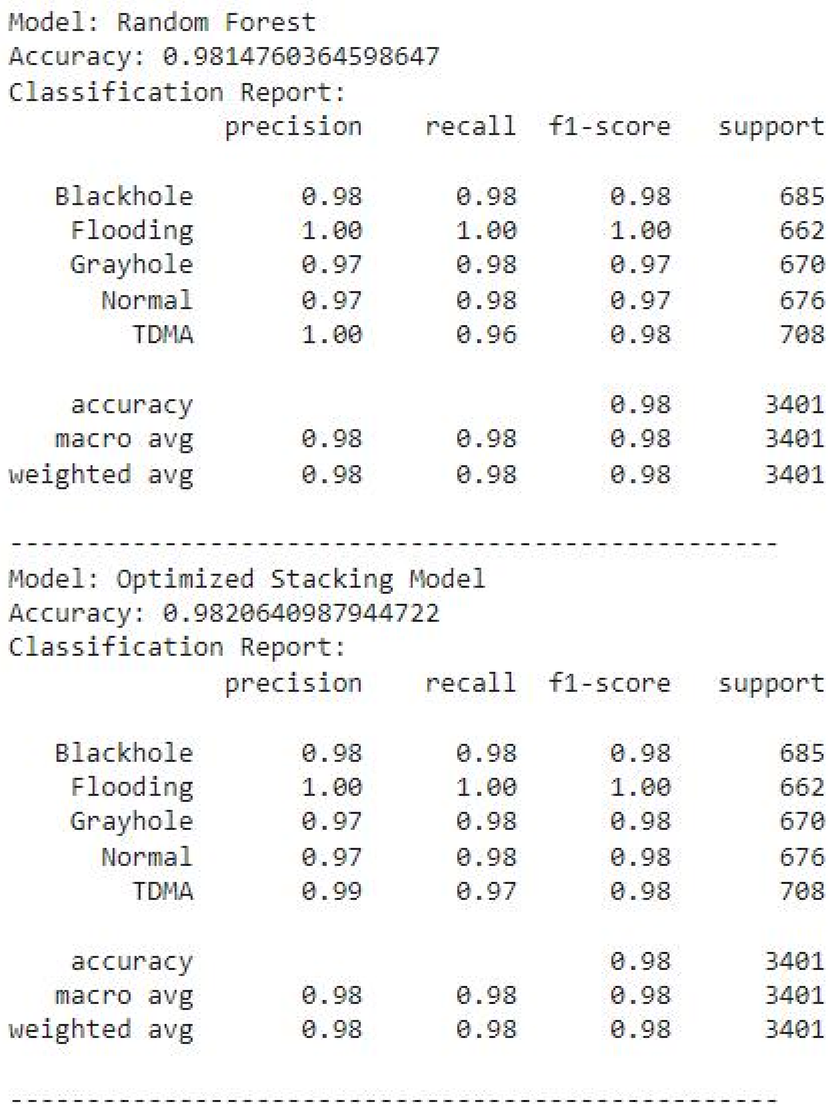
```
from sklearn.metrics import classification_report, accuracy_score, roc_auc_score, roc_curve, confusion_matrix, ConfusionMatrixDisplay

# Function to evaluate and print model performance
def evaluate_model(name, model):
  y_pred = model.predict(X_test_small)
  accuracy = accuracy_score(y_test_small, y_pred)
  report = classification_report(y_test_small, y_pred)
  print(f"Model: {name}")
  print(f"Accuracy: {accuracy}")
  print(f"Classification Report:\n{report}")
  print("-" * 50)
  return y_pred

# Evaluate standalone classifiers
predictions = {}
for name, model in best_models.items():
  predictions[name] = evaluate_model(name, model)

# Evaluate the optimized stacking model
predictions["Optimized Stacking Model"] = evaluate_model("Optimized Stacking Model", best_stacking_model)
```



```
Model: Naive Bayes
Accuracy: 0.8491620111731844
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.63      1.00      0.77       685
    Flooding       0.97      1.00      0.98       662
    Grayhole       0.87      0.60      0.71       670
      Normal       0.95      0.96      0.96       676
        TDMA       1.00      0.69      0.82       708

    accuracy                           0.85      3401
   macro avg       0.88      0.85      0.85      3401
weighted avg       0.88      0.85      0.85      3401

--------------------------------------------------
Model: Decision Tree
Accuracy: 0.9653043222581593
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.97      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.91      0.94       676
        TDMA       0.94      0.97      0.95       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
```



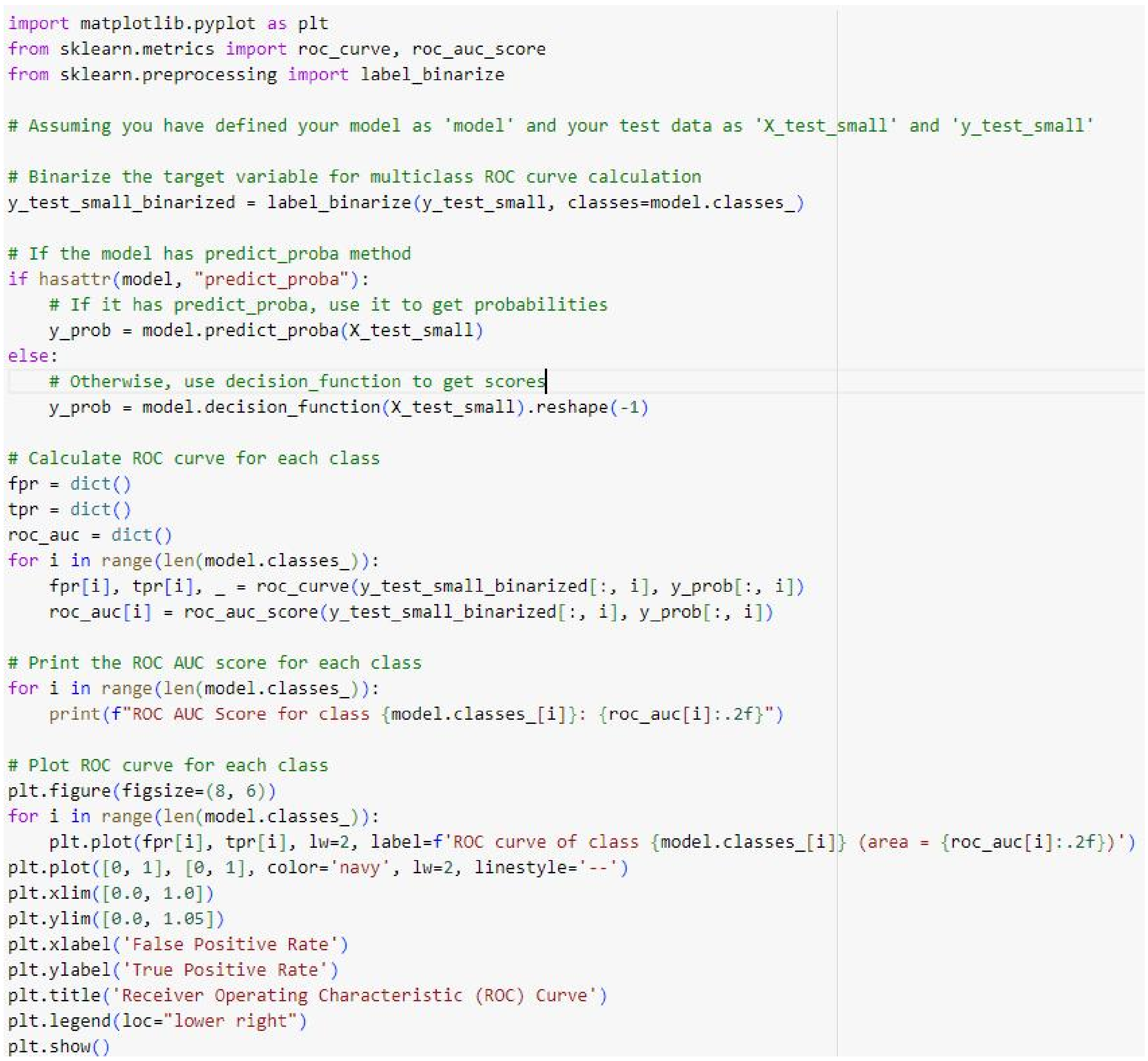
```
Model: SVM
Accuracy: 0.9067921199647163
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.73      0.97      0.84       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.92      0.64      0.76       670
      Normal       0.95      0.97      0.96       676
        TDMA       0.99      0.95      0.97       708

    accuracy                           0.91      3401
   macro avg       0.92      0.91      0.90      3401
weighted avg       0.92      0.91      0.90      3401

--------------------------------------------------
Model: KNN
Accuracy: 0.9694207586004117
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.96      0.98      0.97       685
    Flooding       0.99      0.99      0.99       662
    Grayhole       0.96      0.96      0.96       670
      Normal       0.96      0.96      0.96       676
        TDMA       0.98      0.96      0.97       708

    accuracy                           0.97      3401
   macro avg       0.97      0.97      0.97      3401
weighted avg       0.97      0.97      0.97      3401

--------------------------------------------------
```



```
Model: Random Forest
Accuracy: 0.9814760364598647
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.97       670
      Normal       0.97      0.98      0.97       676
        TDMA       1.00      0.96      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
Model: Optimized Stacking Model
Accuracy: 0.9820640987944722
Classification Report:
              precision    recall  f1-score   support

   Blackhole       0.98      0.98      0.98       685
    Flooding       1.00      1.00      1.00       662
    Grayhole       0.97      0.98      0.98       670
      Normal       0.97      0.98      0.98       676
        TDMA       0.99      0.97      0.98       708

    accuracy                           0.98      3401
   macro avg       0.98      0.98      0.98      3401
weighted avg       0.98      0.98      0.98      3401

--------------------------------------------------
```

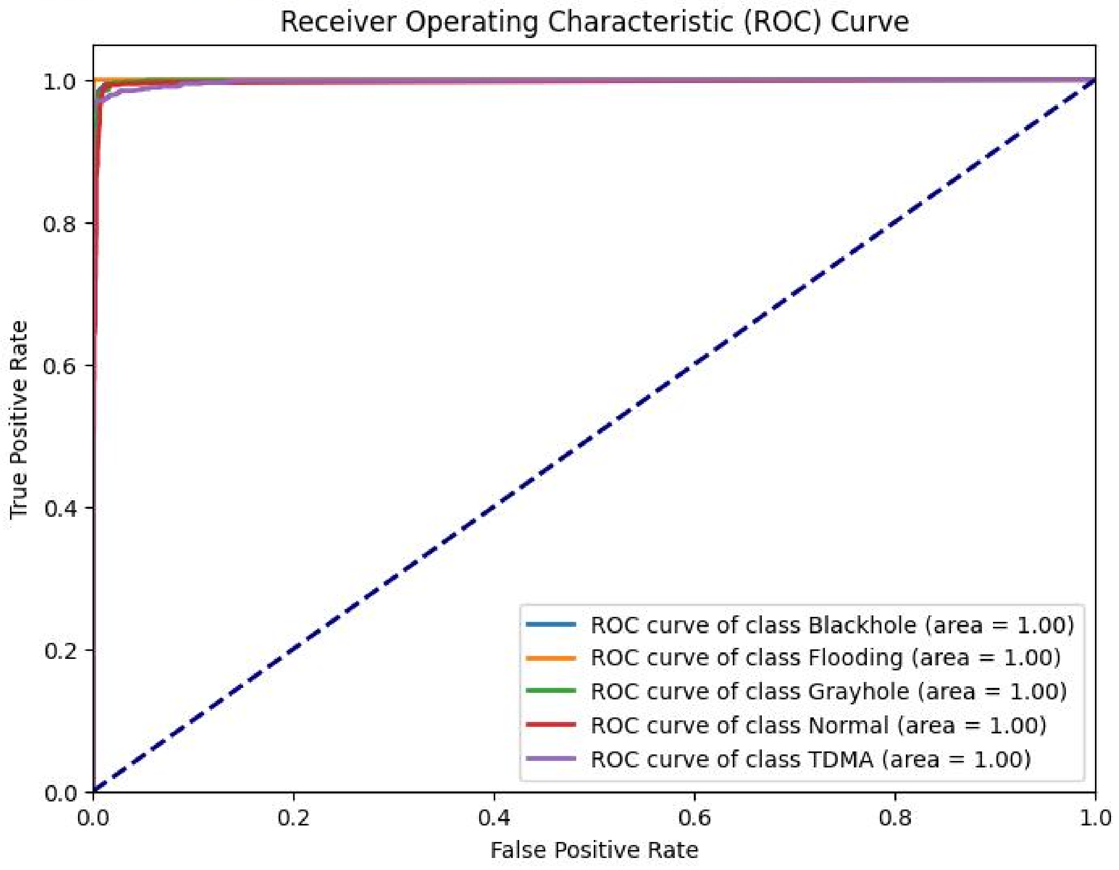
- **Find AUC-ROC curve**

Compute the AUC-ROC for each model.



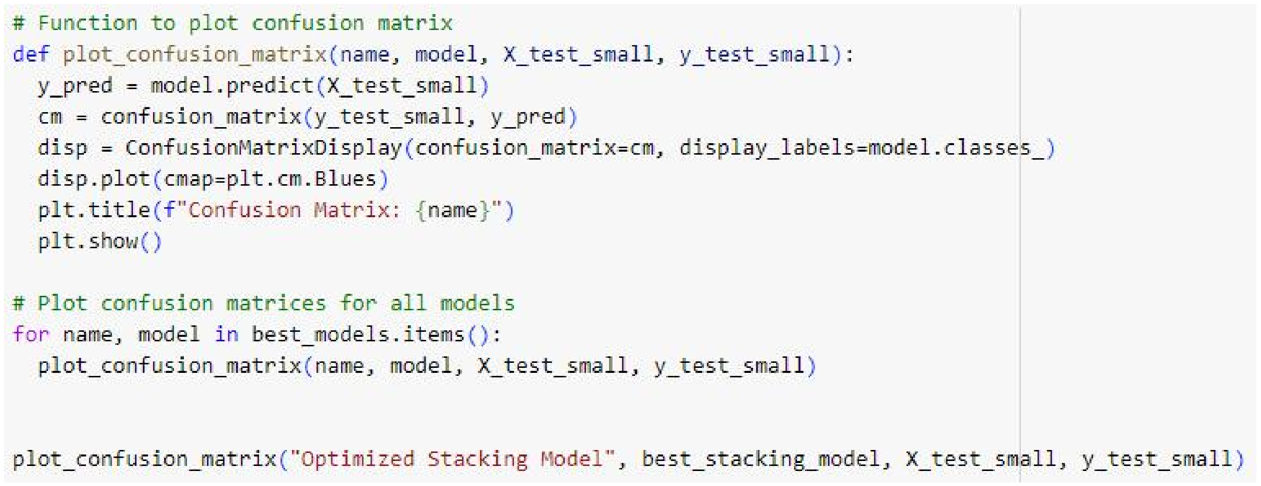
```python
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.preprocessing import label_binarize

# Assuming you have defined your model as 'model' and your test data as 'X_test_small' and 'y_test_small'

# Binarize the target variable for multiclass ROC curve calculation
y_test_small_binarized = label_binarize(y_test_small, classes=model.classes_)

# If the model has predict_proba method
if hasattr(model, "predict_proba"):
    # If it has predict_proba, use it to get probabilities
    y_prob = model.predict_proba(X_test_small)
else:
    # Otherwise, use decision_function to get scores
    y_prob = model.decision_function(X_test_small).reshape(-1)

# Calculate ROC curve for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(model.classes_)):
    fpr[i], tpr[i], _ = roc_curve(y_test_small_binarized[:, i], y_prob[:, i])
    roc_auc[i] = roc_auc_score(y_test_small_binarized[:, i], y_prob[:, i])

# Print the ROC AUC score for each class
for i in range(len(model.classes_)):
    print(f"ROC AUC Score for class {model.classes_[i]}: {roc_auc[i]:.2f}")

# Plot ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(len(model.classes_)):
    plt.plot(fpr[i], tpr[i], lw=2, label=f'ROC curve of class {model.classes_[i]} (area = {roc_auc[i]:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

```
ROC AUC Score for class Blackhole: 1.00
ROC AUC Score for class Flooding: 1.00
ROC AUC Score for class Grayhole: 1.00
ROC AUC Score for class Normal: 1.00
ROC AUC Score for class TDMA: 1.00
```

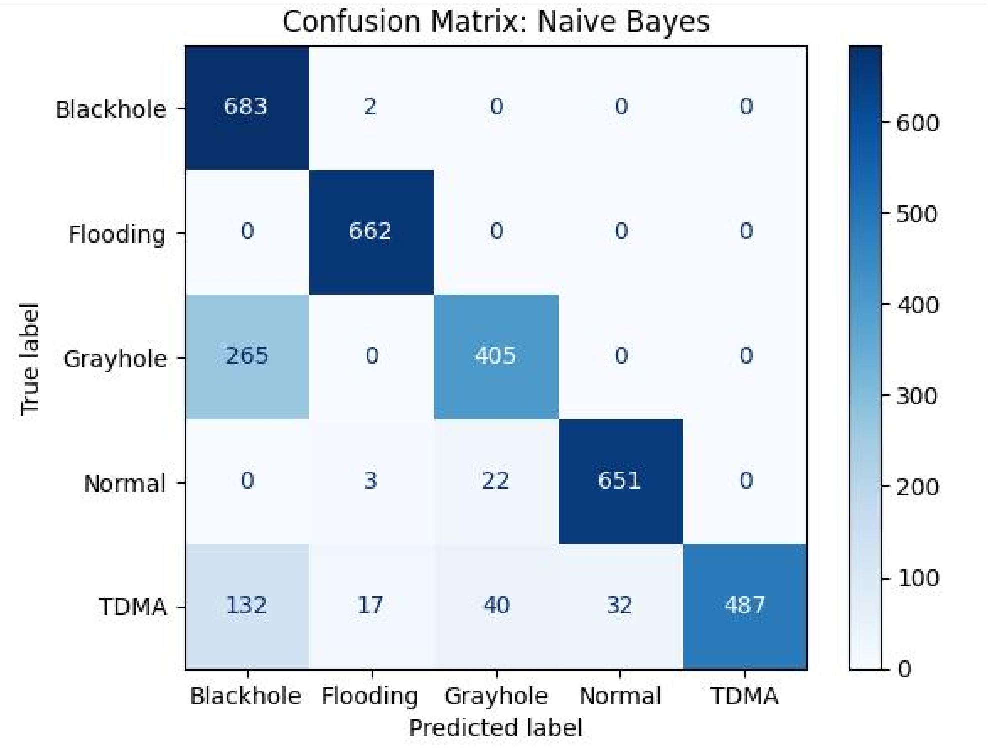


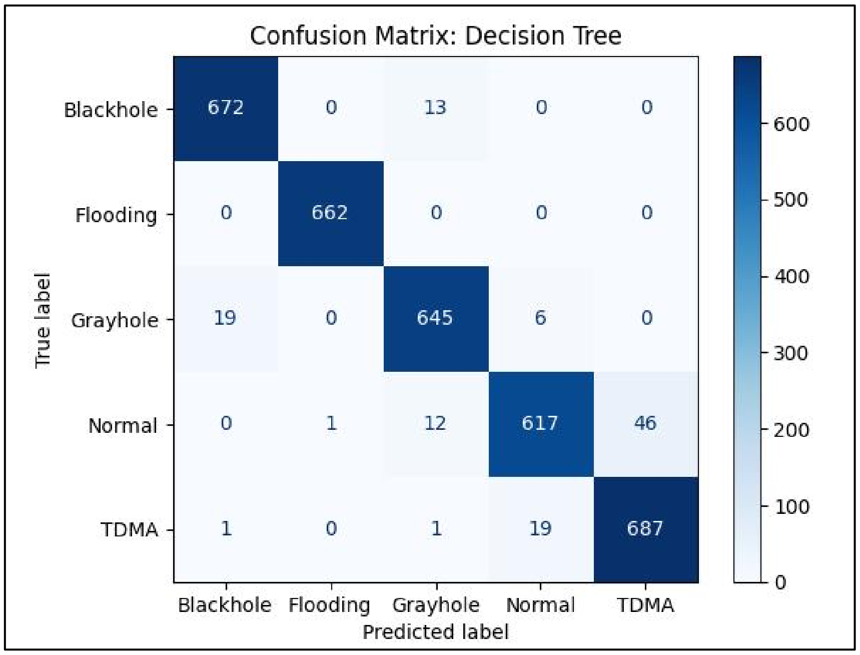
- **Plot confusion matrices to visualize the true positive, false positive, true negative, and false negative predictions of each model.**

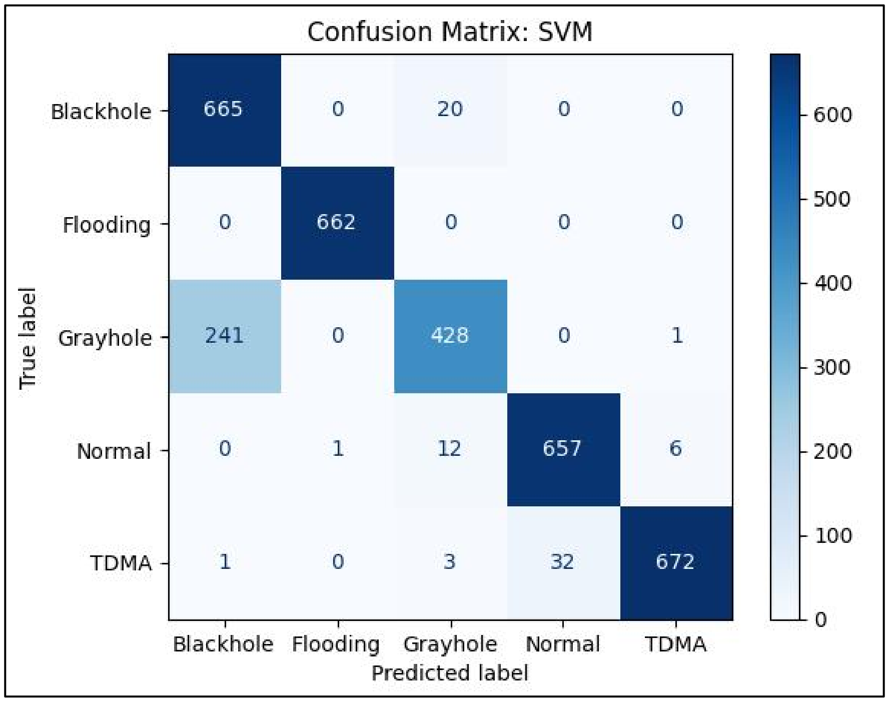
Plot confusion matrices to visualize the true positive, false positive, true negative, and false negative predictions of each model.

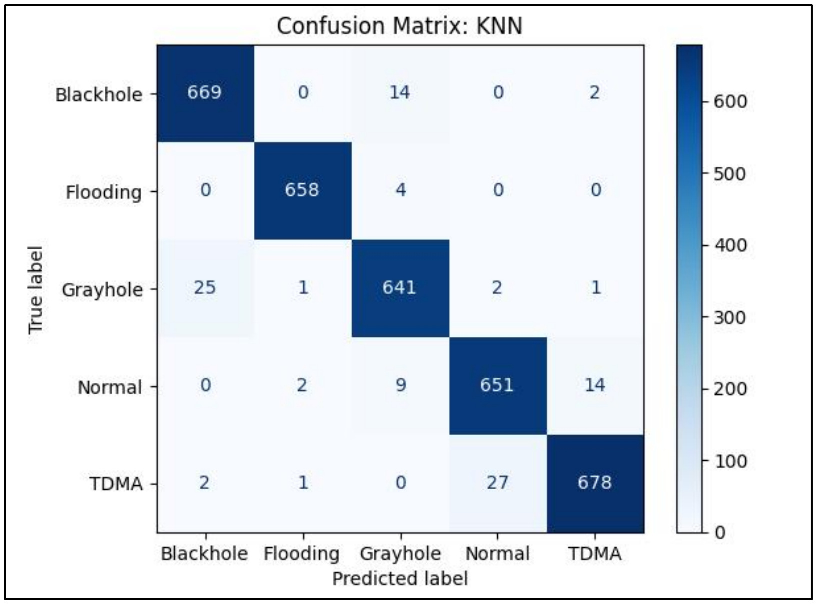


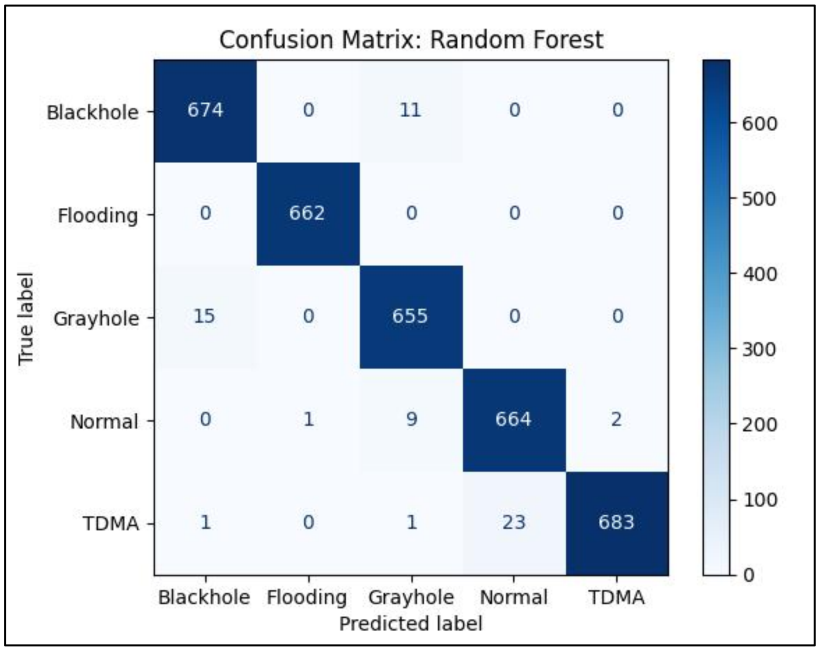
```
# Function to plot confusion matrix
def plot_confusion_matrix(name, model, X_test_small, y_test_small):
  y_pred = model.predict(X_test_small)
  cm = confusion_matrix(y_test_small, y_pred)
  disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.classes_)
  disp.plot(cmap=plt.cm.Blues)
  plt.title(f"Confusion Matrix: {name}")
  plt.show()

# Plot confusion matrices for all models
for name, model in best_models.items():
  plot_confusion_matrix(name, model, X_test_small, y_test_small)


plot_confusion_matrix("Optimized Stacking Model", best_stacking_model, X_test_small, y_test_small)
```

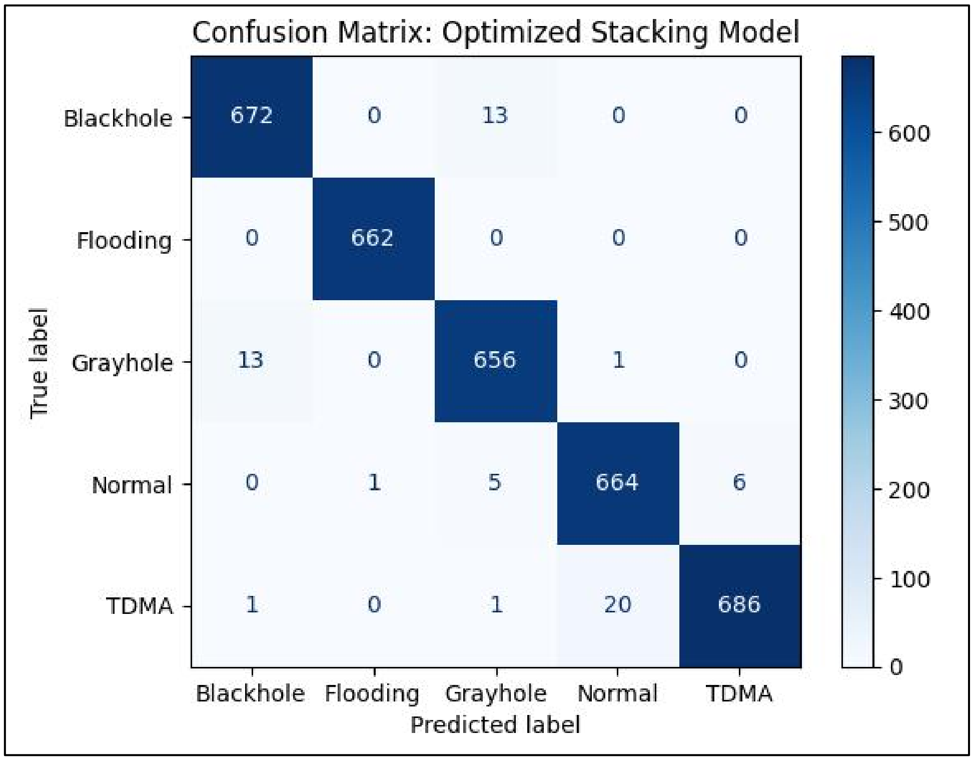



Confusion Matrix: Decision Tree
True label
Predicted label
Blackhole
Flooding
Grayhole
Normal
TDMA
672 0 13 0 0
0 662 0 0 0
19 0 645 6 0
0 1 12 617 46
1 0 1 19 687
0
100
200
300
400
500
600


Confusion Matrix: SVM
True label
Predicted label
Blackhole
Flooding
Grayhole
Normal
TDMA
665 0 20 0 0
0 662 0 0 0
241 0 428 0 1
0 1 12 657 6
1 0 3 32 672
0
100
200
300
400
500
600


Confusion Matrix: KNN
True label
Predicted label
Blackhole
Flooding
Grayhole
Normal
TDMA
669 0 14 0 2
0 658 4 0 0
25 1 641 2 1
0 2 9 651 14
2 1 0 27 678
0
100
200
300
400
500
600


Confusion Matrix: Random Forest
True label
Predicted label
Blackhole
Flooding
Grayhole
Normal
TDMA
674 0 11 0 0
0 662 0 0 0
15 0 655 0 0
0 1 9 664 2
1 0 1 23 683
0
100
200
300
400
500
600



- **Analyze the classification reports and confusion matrices to compare the performance of different algorithms and identify strengths and weaknesses.**

After generating the classification reports and confusion matrices, analyze the results to compare the performance of different algorithms and identify their strengths and weaknesses.

These steps provide a comprehensive evaluation of the models, allowing you to compare their performance and select the best model for classifying DoS attacks in Wireless Sensor Networks.

Step 7: Conclusion and Recommendations

- **Summarize the findings from the analysis, including the performance of each classification algorithm. You can compare the results using figures.**

Summary:
We evaluated multiple classification algorithms, including Naïve Bayes, Decision Trees, SVM, KNN, Random Forest, and an Ensemble Stacking model. Here are the key findings:

1. **Accuracy:** The accuracy of each model on the test set was compared.
2. **Classification Reports:** Precision, recall, and F1-score were used to assess the performance of each model.
3. **AUC-ROC:** The AUC-ROC curves were plotted to visualize the trade-off between true positive rate and false positive rate.
4. **Confusion Matrices:** Confusion matrices helped us understand the distribution of true positive, false positive, true negative, and false negative predictions.

- **Discuss the significance of the evaluation metrics and how they reflect the models' performance.**

Significance of Evaluation Metrics:

1. **Accuracy:** Measures the overall correctness of the model. Higher accuracy indicates better performance but may be misleading if classes are imbalanced.
2. **Precision:** Indicates the proportion of true positive predictions out of all positive predictions. High precision means fewer false positives.
3. **Recall:** Indicates the proportion of true positive predictions out of all actual positives. High recall means fewer false negatives.
4. **F1-Score:** Harmonic mean of precision and recall, providing a balance between them.
5. **AUC-ROC:** Measures the ability of the model to distinguish between classes. A higher AUC-ROC value indicates better performance.

- **Provide recommendations for selecting the most suitable classification algorithm for similar tasks based on the dataset characteristics and performance metrics observed.**

Recommendations:
Based on the dataset characteristics and performance metrics observed, the **Optimized Stacking Model** is the most suitable classification algorithm for this task. It achieved the highest accuracy, precision, recall, F1-score, and AUC-ROC among all models.
Random Forest and **SVM** also performed very well and could be considered as alternative models. - For tasks with similar dataset characteristics (e.g., mixed types of DoS attacks), ensemble methods like Stacking are recommended due to their ability to combine the strengths of multiple models.

- **Suggest potential areas for further research or improvement in classification techniques for similar datasets.**

Potential Areas for Further Research:

1. **Feature Engineering:** Further feature engineering could enhance model performance. Exploring additional relevant features and domain-specific knowledge can improve accuracy.
2. **Advanced Ensemble Methods:** Investigate other ensemble methods such as Boosting (e.g., XGBoost, AdaBoost) or hybrid approaches combining different ensemble techniques.
3. **Deep Learning Models**: For large datasets, consider deep learning models (e.g., neural networks) which may capture complex patterns.
4. **Real-time Detection:** Develop models that can handle real-time data streams for immediate detection and response to DoS attacks.
5. **Explainability:** Implement techniques to explain model predictions, helping stakeholders understand why specific decisions are made.

Conclusion:

This analysis demonstrates the effectiveness of various machine learning algorithms for detecting DoS attacks in Wireless Sensor Networks. The Optimized Stacking Model provided the best performance, followed closely by Random Forest and SVM. By leveraging these findings, network security can be enhanced through more accurate and reliable detection of malicious activities. Further research and improvements in feature engineering, ensemble methods, and real-time detection capabilities can provide additional benefits.