import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from sklearn.preprocessing import StandardScaler

# Phase 1: Data Collection (e.g., UGRansome dataset)

pd.set\_option("expand\_frame\_repr", False)

df= pd.read\_csv('/kaggle/input/ugransome-dataset/final(2).csv')

df2 = pd.DataFrame(df)

df2.columns = ['Time','Protocol','Flag','Family','Clusters','SeedAddress','ExpAddress','BTC','USD','Netflow\_Bytes','IPaddress','Threats','Port','Prediction']

df2

A screen shot of a black screen

Description automatically generated

# Data cleaning

# Renaming the attack "Bonet" to "Botnet"

df2['Threats'] = df2['Threats'].str.replace('Bonet', 'Botnet')

# Print the modified DataFrame

df2

A screen shot of a black screen

Description automatically generated

# Phase 2: Data Preparation (feature engineering and data transformation)

# --- Drop all duplicate rows --- #

df2 = df2.drop\_duplicates()

# --- Remove negative values from time/timestamp feature --- #

df2['Time'] = df2['Time'] + 11

# adding 11 to each value in the 'Time' column of the DataFrame 'df2'.

#In other words, it's performing an element-wise addition operation on all the values in the 'Time' column,

#increasing each value by 11 units. This is often done in data manipulation to shift or adjust time or numerical values

#by a fixed amount

# --- Math transformations to reduce skewness --- #

# --- Log transformation applied to column NETFLOW\_BYTES --- #

# A log transformation involves taking the natural logarithm (base e) of each data point in a particular column or feature.

#Logarithmic transformations are often used to reduce the impact of extreme values (outliers) and make the data conform more

#closely to a normal distribution. They are particularly useful when dealing with positively skewed data,

#where the tail of the distribution is elongated on the right side.

#The np.log() function is a common way to perform a logarithmic transformation in Python.

#The + 1 added to the data points is often used to avoid issues with taking the logarithm of zero or negative values.

#It's a common practice to add a small constant like 1 to the data before applying the logarithm.

#By applying a log transformation to a feature, you're essentially compressing the range of values in that feature,

#which can help in cases where the data exhibits a rightward skew, making it more suitable for certain types of analysis

#or modeling techniques that assume normally distributed data.

df2['Netflow\_Bytes'] = np.log(df2['Netflow\_Bytes']+1)

# --- Square root transformation applied to columns USD ---#

#Square Root Transformation: A square root transformation involves taking the square root of each data point in the

#specified column. In this case, it's applied to the 'USD' column.

#Square root transformations are a type of mathematical transformation used to mitigate the impact of right-skewed data.

#Just like logarithmic transformations, square root transformations can help make the data more symmetric and closer to

#a normal distribution.

#The np.sqrt() function is used to calculate the square root.

#By applying a square root transformation to the 'USD' column, the code is attempting to make the data distribution less skewed

#and more suitable for certain statistical analyses or modeling techniques that assume normally distributed data or

#require data to be more symmetric. It's a common technique used in data preprocessing to improve the quality of data for

#analysis or modeling

df2['USD'] = np.sqrt(df2['USD'])

# --- Yeo Johnson transformation applied to columns BTC--#

#Yeo-Johnson transformation is being applied to the 'BTC' column in the DataFrame (df2['BTC']).

#This transformation is used to modify the data in the 'BTC' column to make its distribution more normalized or symmetric

#The Yeo-Johnson transformation is a mathematical transformation technique used to modify the distribution of data.

#It can be applied to both positive and negative values and is more versatile than some other transformations like the Box-Cox transformation.

#The transformation is performed using the stats.yeojohnson() function from a library like SciPy

df2['BTC'], \_ = stats.yeojohnson(df2['BTC'])

#--PLOTING TRANSFORMED DATA--#

fig, ax = plt.subplots(figsize=(10, 6))

# Plot the transformed 'USD' column

ax.hist(df2['USD'], bins=50, alpha=0.5, color='blue', label='USD (Square Root)')

# Plot the transformed 'BTC' column

ax.hist(df2['BTC'], bins=50, alpha=0.5, color='green', label='BTC (Yeo-Johnson)')

# Plot the transformed 'Netflow\_Bytes' column

ax.hist(df2['Netflow\_Bytes'], bins=50, alpha=0.5, color='red', label='Netflow\_Bytes (Log)')

# Add labels and a legend

ax.set\_xlabel('Transformed Values')

ax.set\_ylabel('Frequency')

ax.set\_title('Distribution of Transformed Columns')

ax.legend()

# Show the plot

plt.show()

# Create a figure and axis for the plot

fig, ax = plt.subplots(figsize=(10, 6))

# Create a StandardScaler instance

# The StandardScaler is a common preprocessing technique used in machine learning and data analysis.

#It is used to standardize or normalize the features of a dataset by scaling them such that they have a mean of 0 and a standard

#deviation of 1.

#Standardizing the features is useful because it makes different features more directly comparable, especially in algorithms

#that are sensitive to the scale of the input data, such as many machine learning algorithms.

#In the code provided, scaler is created as an instance of the StandardScaler class, which can then be used to standardize

#the specified columns in the df2 DataFrame using the fit\_transform method, as seen in the subsequent code

scaler = StandardScaler()

# Normalize each column's features

df2\_normalized = df2.copy()

df2\_normalized[['USD', 'BTC', 'Netflow\_Bytes']] = scaler.fit\_transform(df2[['USD', 'BTC', 'Netflow\_Bytes']])

# Plot the density of the normalized 'USD' column

sns.kdeplot(df2\_normalized['USD'], color='blue', label='USD (Square Root)', ax=ax)

# Plot the density of the normalized 'BTC' column

sns.kdeplot(df2\_normalized['BTC'], color='green', label='BTC (Yeo-Johnson)', ax=ax)

# Plot the density of the normalized 'Netflow\_Bytes' column

sns.kdeplot(df2\_normalized['Netflow\_Bytes'], color='red', label='Netflow\_Bytes (Log)', ax=ax)

# Add labels and a legend

ax.set\_xlabel('Normalized Values')

ax.set\_ylabel('Density')

ax.set\_title('Density Plot of Normalized Columns')

ax.legend()

# Show the plot

plt.show()

A screenshot of a graph

Description automatically generated

# Phase 3: Data Visualization

# --- Count visualizations --- #

# Categorical count visualizations

# Protocol count

ax = sns.countplot(x=df2['Protocol'], data=df2)

plt.title('Bar Graph of Protocol')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Flag count

ax = sns.countplot(x=df2['Flag'], data=df2)

plt.title('Bar Graph of Flag')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Family count

plt.figure(figsize=(15, 6))

ax = sns.countplot(x=df2['Family'], data=df2)

plt.title('Bar Graph of Family')

plt.xticks(rotation=45)

plt.xticks(fontsize=10)

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Clusters count

ax = sns.countplot(x=df2['Clusters'], data=df2)

plt.title('Bar Graph of Clusters')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# SeedAddress count

ax = sns.countplot(x=df2['SeedAddress'], data=df2)

plt.title('Bar Graph of SeedAddress')

plt.xticks(rotation=45)

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# ExpAddress count

ax = sns.countplot(x=df2['ExpAddress'], data=df2)

plt.title('Bar Graph of ExpAddress')

plt.xticks(rotation=45)

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# IPaddress count

ax = sns.countplot(x=df2['IPaddress'], data=df2)

plt.title('Bar Graph of IPaddress')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Threats count

ax = sns.countplot(x=df2['Threats'], data=df2)

plt.title('Bar Graph of Threats')

plt.xticks(rotation=45)

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Port count

ax = sns.countplot(x=df2['Port'], data=df2)

plt.title('Bar Graph of Port')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

# Prediction count

ax = sns.countplot(x=df2['Prediction'], data=df2)

plt.title('Bar Graph of Prediction')

for p in ax.patches:

    ax.annotate(f'{int(p.get\_height())}', (p.get\_x() + p.get\_width() / 2., p.get\_height()),

                ha='center', va='center', fontsize=10, color='black', xytext=(0, 5),

                textcoords='offset points')

plt.show()

A screenshot of a computer screen

Description automatically generated

A graph of a bar chart

Description automatically generated with medium confidence

A graph of different colored bars

Description automatically generated with medium confidence

A bar graph with different colored bars

Description automatically generated

# --- Numeric visualizations (count, mean and standard deviation) --- #

# Time

feature = 'Time'

data = df2[feature]

mean = np.mean(data)

std\_dev = np.std(data)

ax = sns.histplot(data, bins=30, kde=True, color='skyblue', edgecolor='black', alpha=0.7)

ax.lines[0].set\_color('black')

plt.axvline(mean, color='red', linestyle='dashed', linewidth=1, label=f'Mean: {mean:.2f}')

plt.axvline(mean - std\_dev, color='green', linestyle='dashed', linewidth=1, label=f'Mean - Std Dev: {mean - std\_dev:.2f}')

plt.axvline(mean + std\_dev, color='orange', linestyle='dashed', linewidth=1, label=f'Mean + Std Dev: {mean + std\_dev:.2f}')

plt.axvline(std\_dev, color='blue', linestyle='dotted', linewidth=1, label=f'Std Dev: {std\_dev:.2f}')

plt.legend(loc='upper right')

plt.title(f'Histogram of {feature}')

plt.xlabel(feature)

plt.ylabel('Frequency')

plt.show()

# # BTC

feature = 'BTC'

data = df2[feature]

mean = np.mean(data)

std\_dev = np.std(data)

ax = sns.histplot(data, bins=30, kde=True, color='skyblue', edgecolor='black', alpha=0.7)

ax.lines[0].set\_color('black')

plt.axvline(mean, color='red', linestyle='dashed', linewidth=1, label=f'Mean: {mean:.2f}')

plt.axvline(mean - std\_dev, color='green', linestyle='dashed', linewidth=1, label=f'Mean - Std Dev: {mean - std\_dev:.2f}')

plt.axvline(mean + std\_dev, color='orange', linestyle='dashed', linewidth=1, label=f'Mean + Std Dev: {mean + std\_dev:.2f}')

plt.axvline(std\_dev, color='blue', linestyle='dotted', linewidth=1, label=f'Std Dev: {std\_dev:.2f}')

plt.legend(loc='upper right')

plt.title(f'Histogram of {feature}')

plt.xlabel(feature)

plt.ylabel('Frequency')

plt.show()

# # USD

feature = 'USD'

data = df2[feature]

mean = np.mean(data)

std\_dev = np.std(data)

ax = sns.histplot(data, bins=30, kde=True, color='skyblue', edgecolor='black', alpha=0.7)

ax.lines[0].set\_color('black')

plt.axvline(mean, color='red', linestyle='dashed', linewidth=1, label=f'Mean: {mean:.2f}')

plt.axvline(mean - std\_dev, color='green', linestyle='dashed', linewidth=1, label=f'Mean - Std Dev: {mean - std\_dev:.2f}')

plt.axvline(mean + std\_dev, color='orange', linestyle='dashed', linewidth=1, label=f'Mean + Std Dev: {mean + std\_dev:.2f}')

plt.axvline(std\_dev, color='blue', linestyle='dotted', linewidth=1, label=f'Std Dev: {std\_dev:.2f}')

plt.legend(loc='upper right')

plt.title(f'Histogram of {feature}')

plt.xlabel(feature)

plt.ylabel('Frequency')

plt.show()

# Netflow\_Bytes

feature = 'Netflow\_Bytes'

data = df2[feature]

mean = np.mean(data)

std\_dev = np.std(data)

ax = sns.histplot(data, bins=30, kde=True, color='skyblue', edgecolor='black', alpha=0.7)

ax.lines[0].set\_color('black')

plt.axvline(mean, color='red', linestyle='dashed', linewidth=1, label=f'Mean: {mean:.2f}')

plt.axvline(mean - std\_dev, color='green', linestyle='dashed', linewidth=1, label=f'Mean - Std Dev: {mean - std\_dev:.2f}')

plt.axvline(mean + std\_dev, color='orange', linestyle='dashed', linewidth=1, label=f'Mean + Std Dev: {mean + std\_dev:.2f}')

plt.axvline(std\_dev, color='blue', linestyle='dotted', linewidth=1, label=f'Std Dev: {std\_dev:.2f}')

plt.legend(loc='upper right')

plt.title(f'Histogram of {feature}')

plt.xlabel(feature)

plt.ylabel('Frequency')

plt.show()

A screenshot of a graph

Description automatically generated

#The preprocessing module in scikit-learn provides various tools and techniques for preprocessing your data before

#feeding it into machine learning models.

#This preprocessing is crucial to improve the quality of your data and the performance of your models.

from sklearn import preprocessing

#The code segment uses scikit-learn's LabelEncoder to transform categorical variables into numerical values.

#Each categorical column, such as 'Protocol,' 'Flag,' 'Family,' 'SeedAddress,' 'ExpAddress,' 'IPaddress,' 'Threats,' and

#'Prediction,' is encoded into unique numeric labels.

#This preprocessing step is essential for machine learning algorithms, as they typically require numerical input data

#instead of categorical labels.

lab\_encoder = preprocessing.LabelEncoder()                     # transformation of categorical to numeric

df2['Protocol'] = lab\_encoder.fit\_transform(df2['Protocol'])

df2['Flag'] = lab\_encoder.fit\_transform(df2['Flag'])

df2['Family'] = lab\_encoder.fit\_transform(df2['Family'])

df2['SeedAddress'] = lab\_encoder.fit\_transform(df2['SeedAddress'])

df2['ExpAddress'] = lab\_encoder.fit\_transform(df2['ExpAddress'])

df2['IPaddress'] = lab\_encoder.fit\_transform(df2['IPaddress'])

df2['Threats'] = lab\_encoder.fit\_transform(df2['Threats'])

df2['Prediction'] = lab\_encoder.fit\_transform(df2['Prediction'])

df2

A screenshot of a computer

Description automatically generated

#The train\_test\_split function from scikit-learn is used to split a dataset into two subsets:

#a training set and a testing (or validation) set. This function is commonly employed in machine learning

#to assess the performance of a model on unseen data. It takes as input the dataset, typically represented as features (X)

#and labels (y), and divides it into training data (X\_train and y\_train) used to train the model and

#testing data (X\_test and y\_test) used to evaluate the model's performance.

from sklearn.model\_selection import train\_test\_split  # library for machine learning models

#common procedure in machine learning for splitting a dataset into training and testing sets using the train\_test\_split function

#from scikit-learn. Here's a breakdown of what each line of code does:

X = df2.iloc[:, :-1] #This line selects all rows and all columns of the DataFrame df2 except for the last column.

#It's assuming that the last column contains the target variable or labels, and X will contain all the feature columns

y = df2.iloc[:, -1]  # This line selects all rows but only the last column of the DataFrame df2.

#This is to isolate the target variable or labels, and y will contain these labels.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size = 0.8, random\_state = 42)  #split test and train into 4 parts

#This line uses the train\_test\_split function to split the data into training and testing sets.

#Here's a breakdown of the parameters:

#X and y: The feature matrix and target variable.

#train\_size=0.8: This parameter specifies that 80% of the data should be used for training

#(you can adjust this percentage as needed).

#random\_state=42: This parameter sets the random seed for reproducibility, ensuring that the split is the same each time you

#run the code.

#After running this code, you will have:

#X\_train: The feature matrix for training.

#X\_test: The feature matrix for testing.

#y\_train: The target variable for training.

#y\_test: The target variable for testing.

#These subsets can then be used for training and evaluating your machine learning models.

X\_train

X\_test

y\_train

y\_test

A screenshot of a computer

Description automatically generated

#The %%time command is typically used in Jupyter Notebook environments, such as Jupyter Notebook or JupyterLab.

#It is called a "magic command" and is used to measure the execution time of a specific code cell.

#When you include %%time at the beginning of a cell, it tells Jupyter to measure the time it takes to run the code within

#that cell

#%%time

# Import various libraries and tools for building and evaluating machine learning models in Python

# Imported models: ensemble, random forest, SVM, Naive Bayes, genetic algorithm

# Imported evaluation metrics: accuracy, precision, recall, f1 score

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import LinearSVC

from sklearn.naive\_bayes import GaussianNB

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

from sklearn.ensemble import StackingClassifier #ensmbl method of stacking classify for ensmbling

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import classification\_report

from sklearn.tree import DecisionTreeClassifier   #estimator in GA

import numpy as np

import warnings

warnings.filterwarnings('ignore')

rf = RandomForestClassifier(n\_estimators=100, random\_state=42) #  It specifies the number of trees in the Random Forest.

#In this case, there are 100 trees in the forest

# random\_state: This parameter is used to set the random seed for reproducibility.

#By setting it to 42, the randomization process will be the same each time the code is run,

#ensuring consistent results for the Random Forest model.

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

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

#This code snippet uses the trained Random Forest classifier (rf) to make predictions on the test data (X\_test).

#The predict method takes the test features in X\_test as input and produces predicted labels for these features.

#The predictions are stored in the rf\_pred variable, which can be used for further evaluation or analysis to assess how well

#the model performs on unseen data.

rf\_accuracy = accuracy\_score(rf\_pred, y\_test)

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

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

print('Accuracy of Random Forest : ', round(rf\_accuracy, 3))

print('Classification report of Random Forest : \n', rf\_report)

print('Confusion Matrix of Random Forest : \n', rf\_matrix)

#The accuracy\_score function from scikit-learn is used to calculate the accuracy of the model's predictions (rf\_pred)

#compared to the actual labels (y\_test). This score measures the proportion of correctly classified instances.

#classification\_report: The classification\_report function generates a comprehensive report that includes precision, recall,

#F1-score, and support for each class in the classification problem. It provides detailed insights into the model's

#performance for different classes.

#confusion\_matrix: The confusion\_matrix function computes a confusion matrix that summarizes the true positive, true negative,

#false positive, and false negative counts for the classification results. It helps in understanding how well the model is

#performing and where it might be making errors.

#Finally, the code prints out the accuracy, classification report, and confusion matrix for the Random Forest model,

#allowing you to evaluate its performance on the test data.

A screenshot of a computer

Description automatically generated

svr = LinearSVC()

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

svr\_pred = svr.predict(X\_test)

#a Support Vector Machine (SVM) classifier with a linear kernel (LinearSVC) is trained and tested using the following steps:

#svr = LinearSVC(): An instance of the LinearSVC classifier is created.

#svr.fit(X\_train, y\_train): The LinearSVC classifier is trained on the training data (X\_train and y\_train).

#This step involves finding the hyperplane that best separates the data points of different classes while maximizing the margin

#between them.

#svr\_pred = svr.predict(X\_test): The trained SVM classifier is used to make predictions on the test data (X\_test).

#These predictions are stored in the svr\_pred variable.

svr\_accuracy = accuracy\_score(svr\_pred, y\_test)

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

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

print('Accuracy of SVM : ', round(svr\_accuracy, 3))

print('Classification report of SVM : \n', svr\_report)

print('Confusion Matrix of SVM :\n', svr\_matrix)

#svr\_accuracy = accuracy\_score(svr\_pred, y\_test): The accuracy of the SVM classifier's predictions on the test data (svr\_pred)

#is calculated by comparing them to the true labels (y\_test). The result is stored in the svr\_accuracy variable.

#svr\_report = classification\_report(svr\_pred, y\_test): The classification\_report function is used to generate a detailed

#classification report, including metrics such as precision, recall, F1-score, and support for each class.

#This report is stored in the svr\_report variable.

#svr\_matrix = confusion\_matrix(svr\_pred, y\_test): The confusion matrix is computed based on the predictions (svr\_pred) and

#true labels (y\_test). The confusion matrix provides information about the number of true positive, true negative,

#false positive, and false negative predictions. It is stored in the svr\_matrix variable.

#Finally, the results are printed using print statements:

#The accuracy of the SVM classifier is printed with a rounded value.

#The classification report, which includes precision, recall, F1-score, and support for each class, is printed.

#The confusion matrix, which shows the distribution of true and false predictions, is printed.

#These metrics help evaluate the performance of the SVM classifier in terms of its ability to correctly classify data points into different classes.

A screenshot of a computer screen

Description automatically generated

#Naive Bayes Algorithm

nb = GaussianNB()

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

nb\_pred = nb.predict(X\_test)

nb\_accuracy = accuracy\_score(nb\_pred, y\_test)

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

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

print('Accuracy of Naive Bayes : ', round(nb\_accuracy, 3))

print('Classification report of Naive Bayes : \n', nb\_report)

print('Confusion Matrix of Naive Bayes :\n', nb\_matrix)

# Assuming you already have nb\_pred and y\_test defined

nb\_accuracy = accuracy\_score(nb\_pred, y\_test)

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

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

print('Accuracy of Naive Bayes : ', round(nb\_accuracy, 3))

print('Classification report of Naive Bayes : \n', nb\_report)

print('Confusion Matrix of Naive Bayes :\n', nb\_matrix)

# Plot the confusion matrix as a heatmap

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

sns.set(font\_scale=1.2)  # Adjust the font size for better readability

sns.heatmap(nb\_matrix, annot=True, fmt="d", cmap="Blues", cbar=False,

            xticklabels=["0:A", "1:S", "2:SS"], yticklabels=["0:A", "1:S", "2:SS"])

plt.xlabel("Predicted")

plt.ylabel("True")

plt.title("Confusion Matrix Heatmap")

plt.show()

A screenshot of a computer

Description automatically generated

# Assuming you already have nb\_pred and y\_test defined

nb\_accuracy = accuracy\_score(nb\_pred, y\_test)

nb\_report = classification\_report(nb\_pred, y\_test, output\_dict=True)  # Use output\_dict=True to get metrics as a dictionary

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

# Extract support for all classes

labels = [str(label) for label in np.unique(np.concatenate((nb\_pred, y\_test)))]  # Get all unique labels

support = [nb\_report[label]['support'] if label in nb\_report else 0 for label in labels]

print('Accuracy of Naive Bayes : ', round(nb\_accuracy, 3))

print('Classification report of Naive Bayes : \n', classification\_report(nb\_pred, y\_test))

print('Confusion Matrix of Naive Bayes :\n', nb\_matrix)

# Plot support

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

plt.bar(labels, support, width=0.2, label='Support', align='center')

plt.xlabel('Class')

plt.ylabel('Number of Features')

plt.xticks(labels)

plt.legend()

plt.title('Support for Each Class')

plt.show()

A screenshot of a computer

Description automatically generated

# Assuming you already have nb\_pred and y\_test defined

nb\_accuracy = accuracy\_score(nb\_pred, y\_test)

nb\_report = classification\_report(nb\_pred, y\_test, output\_dict=True)  # Use output\_dict=True to get metrics as a dictionary

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

# Extract precision and recall for all classes

labels = [str(label) for label in np.unique(np.concatenate((nb\_pred, y\_test)))]  # Get all unique labels

precision = [nb\_report[label]['precision'] if label in nb\_report else 0.0 for label in labels]

recall = [nb\_report[label]['recall'] if label in nb\_report else 0.0 for label in labels]

print('Accuracy of Naive Bayes : ', round(nb\_accuracy, 3))

print('Classification report of Naive Bayes : \n', classification\_report(nb\_pred, y\_test))

# Plot precision and recall

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

plt.bar(labels, precision, width=0.2, label='Precision', align='center')

plt.bar(labels, recall, width=0.2, label='Recall', align='edge')

plt.xlabel('Class')

plt.ylabel('Score')

plt.xticks(labels)

plt.legend()

plt.title('Precision and Recall for Each Class')

plt.show()

A screenshot of a graph

Description automatically generated

estimators = [

    ('rf', RandomForestClassifier(n\_estimators=1000, random\_state=42)),

    ('svr', LinearSVC(random\_state=42))

]

clf = StackingClassifier(

    estimators=estimators, final\_estimator=GaussianNB())

#In this code snippet, a Stacking Classifier (clf) is defined using scikit-learn's StackingClassifier.

#Stacking is an ensemble learning method that combines multiple base estimators to improve predictive performance.

#Here's a breakdown of the code:

#estimators: This is a list of tuples, where each tuple contains the name of the estimator and the estimator object.

#Two base estimators are defined:

#'rf': A Random Forest Classifier with 1000 estimators and a random seed of 42.

#'svr': A Linear Support Vector Classifier (LinearSVC) with a random seed of 42.

#clf: The Stacking Classifier is created using the StackingClassifier class. It takes two main parameters:

#estimators: This parameter receives the list of base estimators defined earlier.

#final\_estimator: This parameter specifies the meta-estimator that combines the predictions from the base estimators.

#In this case, a Gaussian Naive Bayes (GaussianNB) classifier is used as the final estimator.

#The Stacking Classifier combines the predictions of the base classifiers ('rf' and 'svr') using the final estimator (GaussianNB)

#This ensemble method can often improve classification performance by leveraging the strengths of multiple classifiers.

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

pred = clf.predict(X\_test)

accuracy = accuracy\_score(pred, y\_test)

#In this code snippet, the Stacking Classifier (clf) is trained on the training data (X\_train and y\_train) using the fit method.

#After training, the classifier is used to make predictions on the test data (X\_test) using the predict method.

#Finally, the accuracy of the predictions is calculated using scikit-learn's accuracy\_score function and stored in the variable accuracy.

#The code essentially performs the following steps:

#Trains the Stacking Classifier (clf) using the training data.

#Uses the trained classifier to predict the target labels for the test data.

#Calculates the accuracy of the predictions by comparing them to the true labels (y\_test).

#The accuracy variable will contain the accuracy score of the Stacking Classifier's predictions on the test data.

#This score measures how well the classifier performed in terms of correctly classifying the data points in the test set.

eb\_accuracy = accuracy\_score(pred, y\_test)

eb\_matrix = confusion\_matrix(pred, y\_test)

eb\_report = classification\_report(pred, y\_test)

print('Accuracy of Ensemble Model : ', round(eb\_accuracy, 3))

print('Confusion Matrix of Ensemble Model : ', eb\_matrix)

print('Classification Report of Ensemble Model :', eb\_report)

#In this code snippet, the accuracy, confusion matrix, and classification report for the ensemble model (clf) are evaluated

#and printed.

#Here's what each part of the code does:

#eb\_accuracy: Calculates the accuracy of the ensemble model's predictions by comparing them to the true labels (y\_test)

#using the accuracy\_score function.

#eb\_matrix: Computes the confusion matrix for the ensemble model's predictions using the confusion\_matrix function.

#The confusion matrix provides information about the true positives, true negatives, false positives, and false negatives.

#eb\_report: Generates a classification report for the ensemble model's predictions using the classification\_report function.

#The classification report includes metrics such as precision, recall, F1-score, and support for each class.

#Finally, the code prints out the accuracy, confusion matrix, and classification report for the ensemble model.

#These metrics provide insights into the model's performance in terms of classification accuracy and the ability to correctly

#classify different classes.

# Assuming you already have pred and y\_test defined for your Ensemble Model

eb\_accuracy = accuracy\_score(pred, y\_test)

eb\_matrix = confusion\_matrix(pred, y\_test)

eb\_report = classification\_report(pred, y\_test)

print('Accuracy of Ensemble Model : ', round(eb\_accuracy, 3))

print('Confusion Matrix of Ensemble Model : \n', eb\_matrix)

print('Classification Report of Ensemble Model :\n', eb\_report)

# Plot the confusion matrix as a heatmap

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

sns.set(font\_scale=1.2)  # Adjust the font size for better readability

sns.heatmap(eb\_matrix, annot=True, fmt="d", cmap="Blues", cbar=False,

            xticklabels=["0:A", "1:S", "2:SS"], yticklabels=["0:A", "1:S", "2:SS"])

plt.xlabel("Predicted")

plt.ylabel("True")

plt.title("Confusion Matrix Heatmap for Ensemble Model")

plt.show()

A screenshot of a computer

Description automatically generated

#Plot the evaluation metrics of each model in one figure

# Model names

models = ['Random Forest', 'SVM', 'Naive Bayes', 'Ensemble Learning']

# Precision scores

precision = [100, 82, 64, 99]

# Recall scores

recall = [100, 47, 66, 99]

# F1-score scores

f1\_score = [100, 65, 77, 99]

# X-axis values (models)

x = range(len(models))

# Create a figure and axis for the plot

fig, ax = plt.subplots(figsize=(10, 6))

# Plot precision scores

ax.plot(x, precision, marker='o', linestyle='-', color='b', label='Precision')

# Plot recall scores

ax.plot(x, recall, marker='o', linestyle='-', color='g', label='Recall')

# Plot F1-score scores

ax.plot(x, f1\_score, marker='o', linestyle='-', color='r', label='F1-Score')

# Set x-axis ticks and labels

ax.set\_xticks(x)

ax.set\_xticklabels(models, rotation=45)

ax.set\_xlabel('Machine Learning Models')

# Set y-axis label

ax.set\_ylabel('Scores (%)')

# Set plot title

ax.set\_title('Fluctuation of Precision, Recall, and F1-Score for Different Models')

# Add a legend

ax.legend()

# Show the plot

plt.tight\_layout()

plt.grid(True)

plt.show()

A graph showing the difference between different models

Description automatically generated with medium confidence

# Define the algorithms and their corresponding accuracies

algorithms = ['Random Forest', 'SVM', 'NB', 'Ensemble Learning']

accuracies = [100, 64, 77, 99]

# Create a bar graph

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

plt.bar(algorithms, accuracies, color=['blue', 'red', 'green', 'purple'])

plt.ylim(0, 110)  # Set the y-axis limit for better visualization

plt.xlabel('Algorithms')

plt.ylabel('Accuracy (%)')

plt.title('Accuracy of Different Machine Learning Algorithms')

plt.grid(axis='y', linestyle='--', alpha=0.7)

# Display the accuracy values on top of the bars

for i, v in enumerate(accuracies):

    plt.text(i, v + 2, str(v) + '%', ha='center', va='bottom', fontsize=12)

# Show the graph

plt.tight\_layout()

plt.show()

A graph of different colored squares

Description automatically generated