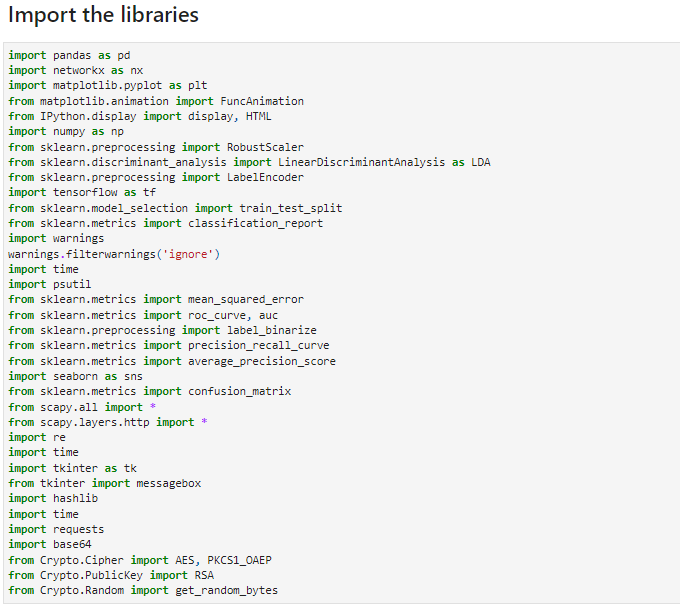
**Cooperative intrusion detection framework using the blockchain**

**Import the Libraries:**



**Explanation:**

* **pandas:** A data manipulation and analysis library for Python, offering data structures and operations for manipulating numerical tables and time series.
* **networkx:** A Python library for the creation, manipulation, and study of complex networks or graphs.
* **matplotlib.pyplot:** A plotting library for Python, providing a MATLAB-like interface for creating static, interactive, and animated visualizations.
* **FuncAnimation (from matplotlib.animation):** A function in Matplotlib for creating animations by repeatedly calling a function to update the plot.
* **IPython.display:** A module providing tools for displaying rich content in the IPython environment, such as HTML, images, and videos.
* **numpy:** A fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
* **sklearn.preprocessing.RobustScaler:** A preprocessing technique for scaling features to robustly handle outliers.
* **sklearn.discriminant\_analysis.LinearDiscriminantAnalysis:** A classification algorithm for dimensionality reduction and linear classification.
* **LabelEncoder (from sklearn.preprocessing):** A utility class for encoding categorical features as numeric labels.
* **tensorflow:** An open-source machine learning framework for building and training machine learning models.
* **train\_test\_split (from sklearn.model\_selection):** A function for splitting datasets into training and testing subsets.
* **classification\_report (from sklearn.metrics):** A function to generate a text report showing the main classification metrics.
* **warnings:** A module for handling warnings in Python.
* **time:** A module providing various time-related functions.
* **psutil:** A cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network).
* **mean\_squared\_error (from sklearn.metrics):** A function for computing the mean squared error regression loss.
* **roc\_curve, auc (from sklearn.metrics):** Functions for computing Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) score.
* **precision\_recall\_curve (from sklearn.metrics):** A function for computing precision-recall curve.
* **average\_precision\_score (from sklearn.metrics):** A function for computing average precision from prediction scores.
* **seaborn:** A Python visualization library based on matplotlib, providing a high-level interface for drawing attractive statistical graphics.
* **confusion\_matrix (from sklearn.metrics):** A function for computing confusion matrix to evaluate the accuracy of a classification.
* **scapy:** A Python library for packet manipulation and network scanning.
* **re:** The regular expression module in Python for working with regular expressions.
* **tkinter:** The standard Python interface to the Tk GUI toolkit for creating desktop applications with graphical user interfaces (GUIs).
* **messagebox (from tkinter):** A module providing a simple and flexible way to display dialog boxes in Tkinter applications.
* **hashlib:** A module providing cryptographic hash functions.
* **requests:** A Python HTTP library for making HTTP requests.
* **base64:** A module providing functions for encoding and decoding data in base64 format.
* **Crypto.Cipher.AES, Crypto.Cipher.PKCS1\_OAEP (from Crypto.Cipher):** Modules for encryption and decryption using the AES and RSA algorithms, respectively.
* **Crypto.PublicKey.RSA (from Crypto.PublicKey):** A module for generating RSA key pairs and managing RSA keys.

**Load the Dataset:**



**Explanation:**

**Reading Data:**

* Reads a CSV file located at the specified path into a pandas DataFrame.

**Filtering by Conditions:**

* Filters the DataFrame based on specific conditions like the number of forward and backward packets and the label.

**Defining Desired Ports and Labels:**

* Creates a dictionary where each key represents a port number, and its corresponding value is a list of labels.

**Filtering Based on Ports and Labels:**

* Iterates over the desired ports and labels, filtering the data based on each combination.
* Filters records where the total number of forward packets is not equal to the total number of backward packets.

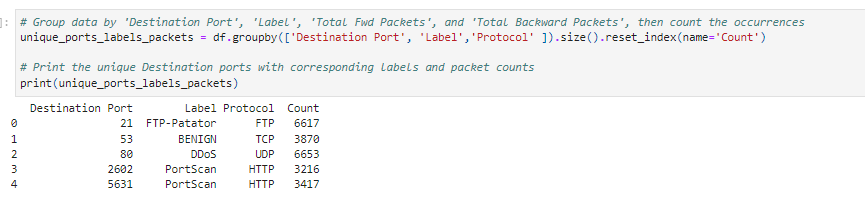
**Concatenating DataFrames:**

* Combines all filtered DataFrames into a single DataFrame using pd.concat().

**Printing the Concatenated DataFrame:**

* Displays the first few rows of the concatenated DataFrame using df.head().

**Group the Data for understanding**



**Explanation:**

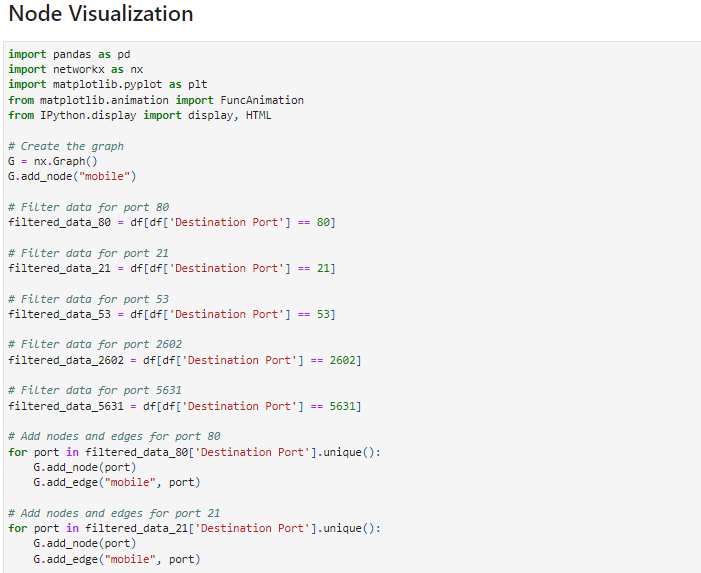
**Grouping Data:**

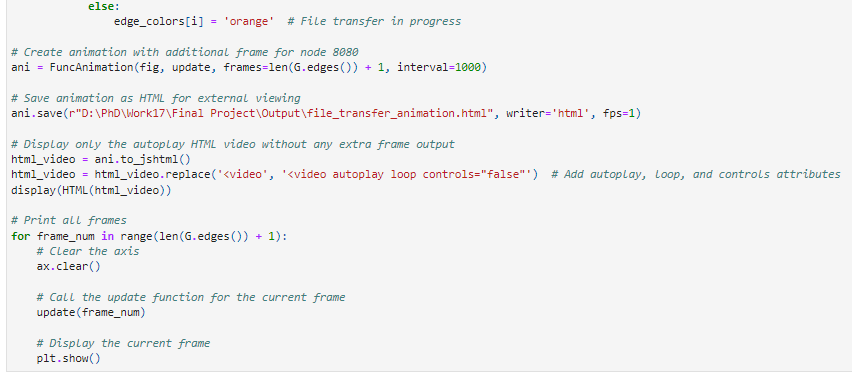
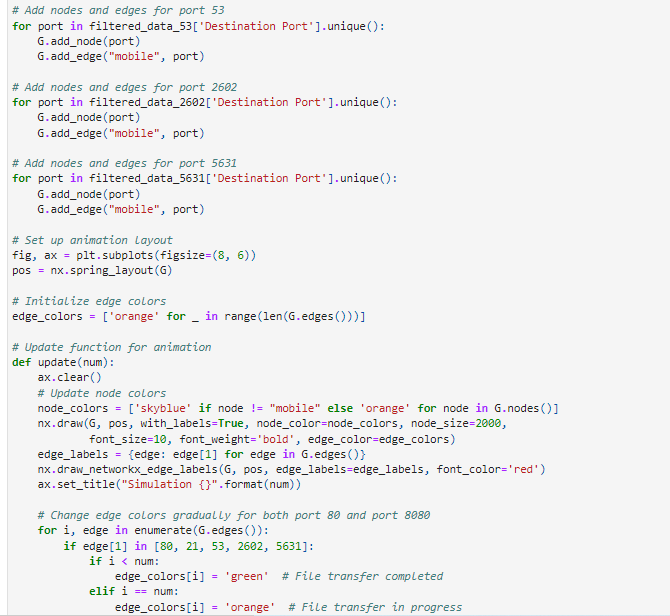
* Groups the DataFrame df by 'Destination Port', 'Label', and 'Protocol', then counts the occurrences of each group.

**Printing Unique Ports with Labels and Packet Counts:**

* Displays the unique combinations of 'Destination Port', 'Label', and 'Protocol', along with the count of occurrences for each combination.

**Node Visualization**





**Explanation:**

**Importing Libraries:**

* Importing necessary libraries including pandas, NetworkX, Matplotlib, and FuncAnimation for animation.

**Creating Graph:**

* Initializes a graph object G using NetworkX and adds a node labeled "mobile" to it.

**Filtering Data:**

* Filters the DataFrame df based on different destination ports (80, 21, 53, 2602, 5631).

**Adding Nodes and Edges:**

* Adds nodes and edges to the graph for each filtered port data, connecting them to the "mobile" node.

**Setting Up Animation Layout:**

* Sets up the layout for the animation by creating a Matplotlib figure and axis.

**Updating Function:**

* Defines an update function for the animation that clears the axis, updates node colors, draws the graph, and sets the title.

**Creating Animation:**

* Creates an animation using FuncAnimation, updating the graph for each frame with changing edge colors to represent file transfer status.

**Saving Animation:**

* Saves the animation as an HTML file for external viewing.

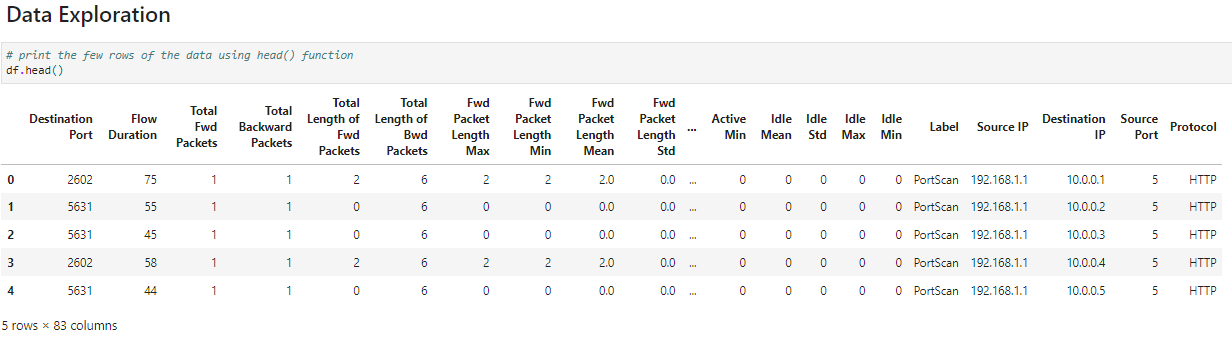
**Displaying HTML Video:**

* Displays the autoplay HTML video without any extra frame output using IPython's display module.

**Printing Frames:**

* Prints all frames of the animation individually.

**Data Exploration:**



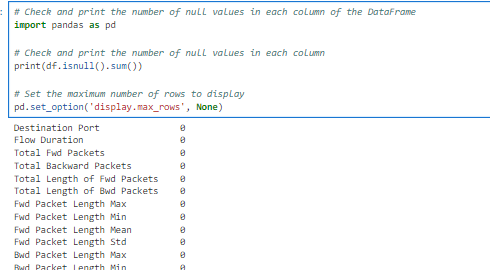
**Head Function Explanation:**

The head() function in pandas is used to display the first few rows of a DataFrame. By default, it displays the first 5 rows.

**Output:**

* The first column contains the index of the DataFrame.
* The subsequent columns represent different features or attributes of the data.
* Each row corresponds to a specific observation or data point.
* The values in the cells represent the values of those attributes for each observation.

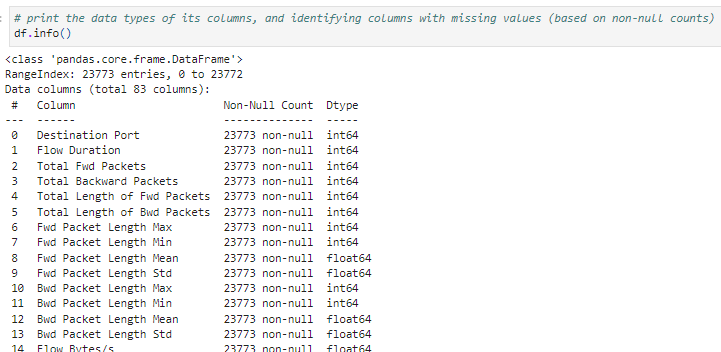
**isnull Function (find the Null values)**



**Explanation:**

* The code checks for and prints the number of null values in each column of the DataFrame. It utilizes the isnull() function to identify null values in the DataFrame, followed by the sum() function to count the number of null values in each column.
* The output displays the number of null values for each column. In this case, there are no null values (NaN) present in any column of the DataFrame. Each column shows a count of 0 null values.

**Information about Dataset:**



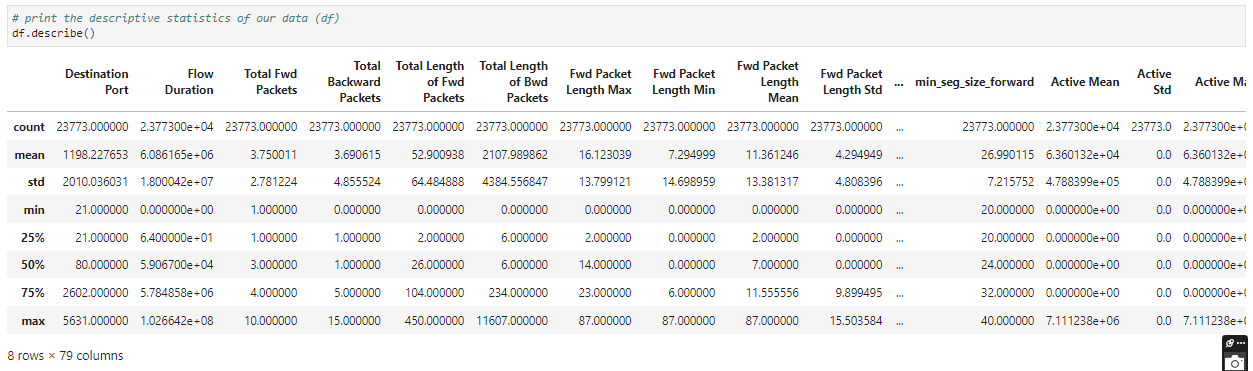
**Explanation:**

The code prints the data types of each column in the DataFrame and also identifies columns with missing values based on non-null counts.

* The DataFrame consists of 23773 entries (rows).
* It contains a total of 82 columns.
* Each column has a non-null count equal to the total number of entries, indicating that there are no missing values in any column.
* The data types of the columns include:
* 20 columns are of type float64.
* 59 columns are of type int64.
* 4 columns are of type object.

The 'Flow Bytes/s' and 'Flow Packets/s' columns are of type object, which might require further investigation to ensure proper data type conversion if they are supposed to contain numeric values.

**Descriptive Statistics:**



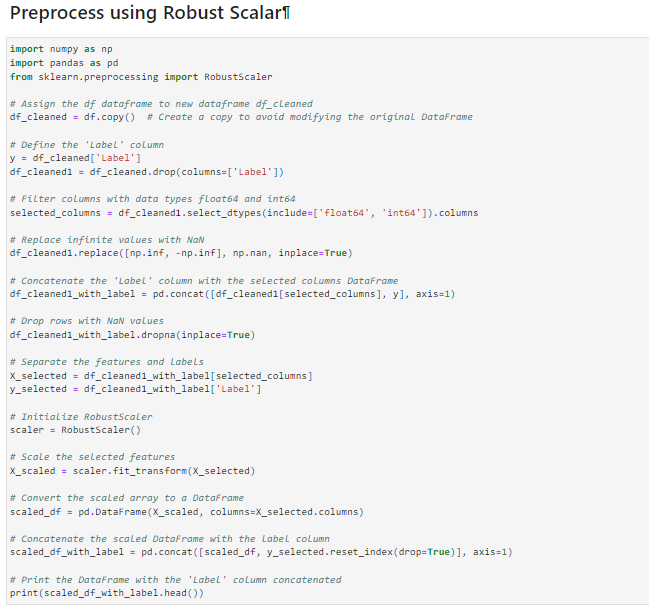
**Explanation:**

The code prints the descriptive statistics of the DataFrame.

* The count row displays the number of non-null values in each column.
* The mean row displays the mean (average) value of each column.
* The std row displays the standard deviation, which measures the dispersion or spread of the values in each column.
* The min row displays the minimum value observed in each column.
* The 25%, 50%, and 75% rows represent the first quartile (25th percentile), median (50th percentile), and third quartile (75th percentile) respectively. They provide information about the distribution of values in each column.
* The max row displays the maximum value observed in each column.

Overall, these descriptive statistics offer insights into the central tendency, variability, and distribution of the data across different columns of the DataFrame. It helps in understanding the range and distribution of values, as well as identifying potential outliers or unusual patterns.

**Preprocess using Robust Scalar**



**Explanation:**

**Data Copying:**

* A copy of the original DataFrame df is made to df\_cleaned to avoid modifying the original data.

**Defining Labels:**

* The 'Label' column is assigned to y, and it is removed from df\_cleaned to obtain df\_cleaned1.

**Selecting Numerical Columns:**

* Columns with data types 'float64' and 'int64' are selected.

**Handling Infinite Values:**

* Infinite values in the DataFrame are replaced with NaN.

**Concatenating with Labels:**

* The 'Label' column is concatenated with the selected numerical columns.

**Handling Missing Values:**

* Rows with NaN values are dropped from the DataFrame.

**Separating Features and Labels:**

* Features (X\_selected) and labels (y\_selected) are separated.

**Initializing RobustScaler:**

* A RobustScaler object is initialized.

**Scaling Features:**

* The selected features are scaled using RobustScaler.

**Converting to DataFrame:**

* The scaled array is converted back to a DataFrame (scaled\_df).

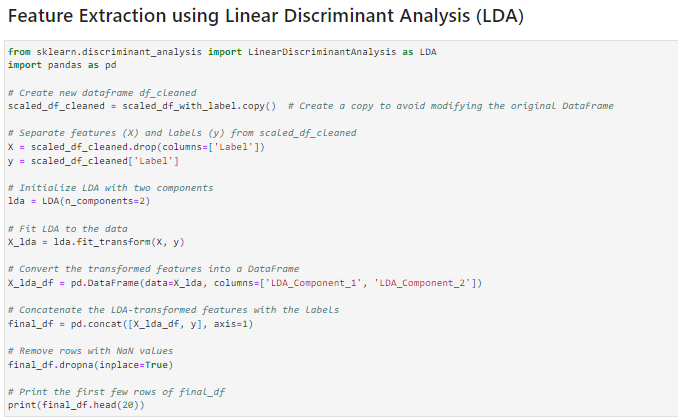
**Concatenating with Labels:**

* The scaled DataFrame is concatenated with the label column (scaled\_df\_with\_label).

**Printing Results:**

* The resulting DataFrame with scaled features and labels is printed.

**Feature Extraction using Linear Discriminant Analysis**



**Explanation:**

**Importing Libraries:**

* The code imports necessary libraries including pandas and LinearDiscriminantAnalysis from sklearn.

**Creating a Copy:**

* A copy of the DataFrame scaled\_df\_with\_label is created to scaled\_df\_cleaned to avoid modifying the original data.

**Separating Features and Labels:**

* Features (X) and labels (y) are separated from the cleaned DataFrame.

**Initializing LDA:**

* LDA is initialized with two components.

**Fitting LDA:**

* LDA is fitted to the data to find the linear discriminants that best separate the classes.

**Transforming Features:**

* The features are transformed using the fitted LDA model to reduce their dimensionality to two components.

**Converting to DataFrame:**

* The transformed features are converted into a DataFrame X\_lda\_df.

**Concatenating with Labels:**

* The LDA-transformed features are concatenated with the labels to form the final DataFrame final\_df.

**Handling Missing Values:**

* Rows with NaN values are removed from the final DataFrame.

**Printing Results:**

* The first few rows of the final DataFrame are printed to inspect the results.

**Label Encode**



**Explanation:**

**Importing Libraries:**

* The code imports the LabelEncoder from sklearn.preprocessing.

**Defining Label Encoding Dictionary:**

* A dictionary label\_encoding is defined to map class labels to numerical values.

**Applying Label Encoding:**

* Label encoding is applied to the 'Label' column of final\_df using the defined label encoding dictionary.

**Handling Non-finite Values:**

* Rows with non-finite values in the 'Label' column are dropped from the DataFrame to ensure consistency.

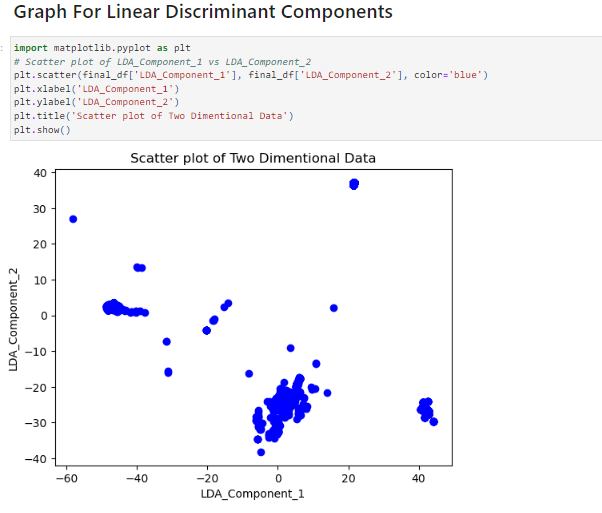
**Converting Data Type:**

* The data type of the 'Label' column is converted to integer.

**Printing Results:**

* The first few rows of the DataFrame with encoded labels are printed for inspection.

**Graph For Linear Discriminant Components**



**Explanation:**

Scatter Plot of LDA Components: The script utilizes matplotlib.pyplot to create a scatter plot of the extracted LDA components (LDA\_Component\_1 and LDA\_Component\_2).

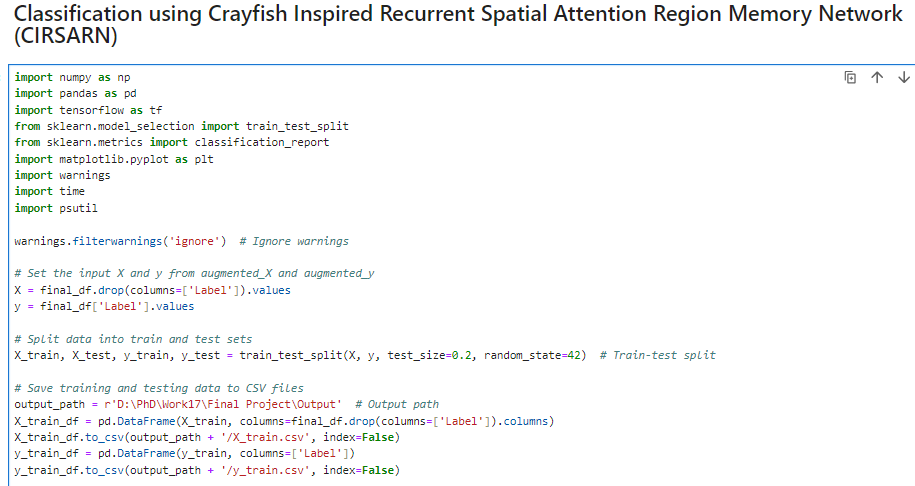
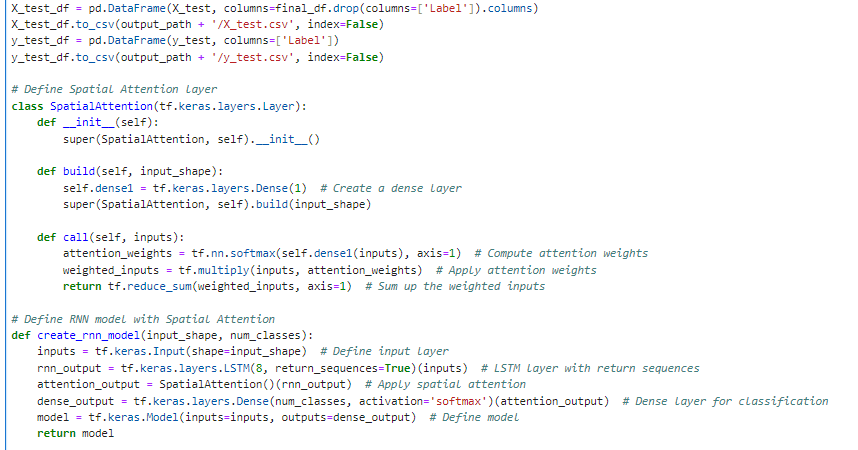
* plt.scatter: The scatter plot is created with LDA\_Component\_1 on the x-axis and LDA\_Component\_2 on the y-axis. Data points are represented as blue dots.
* plt.xlabel: The x-axis label is set as 'LDA\_Component\_1'.
* plt.ylabel: The y-axis label is set as 'LDA\_Component\_2'.
* plt.title: The title of the plot is set as 'Scatter plot of Two Dimensional Data'.
* plt.show: Finally, the plot is displayed.

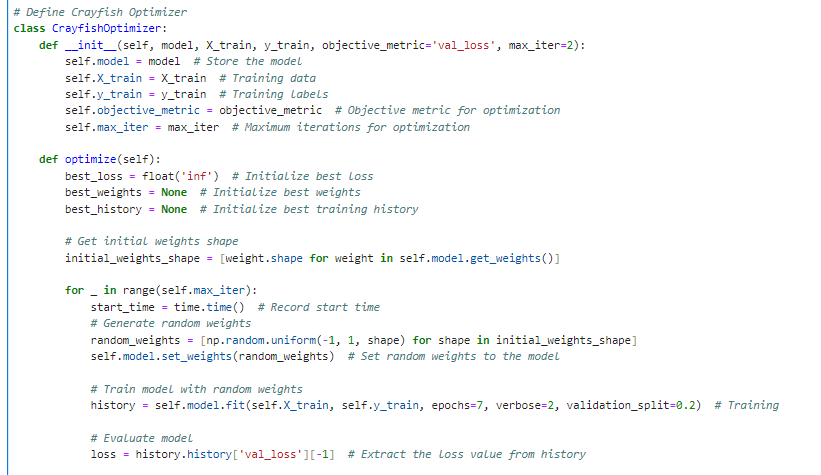
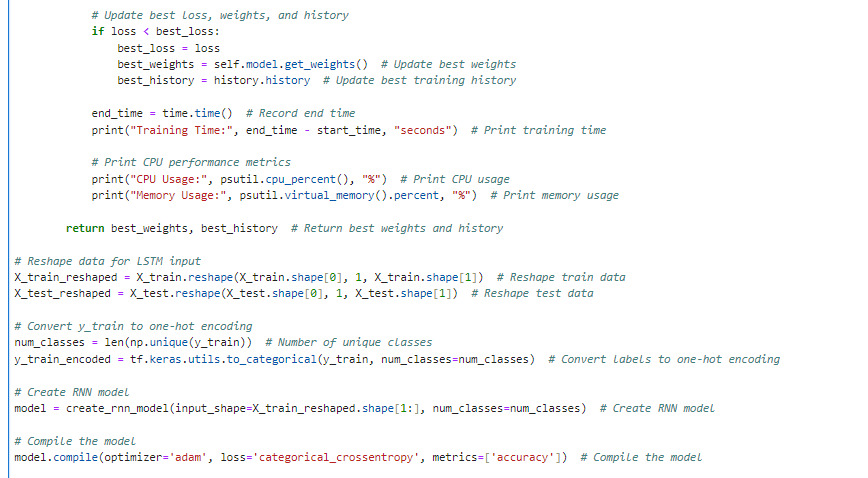
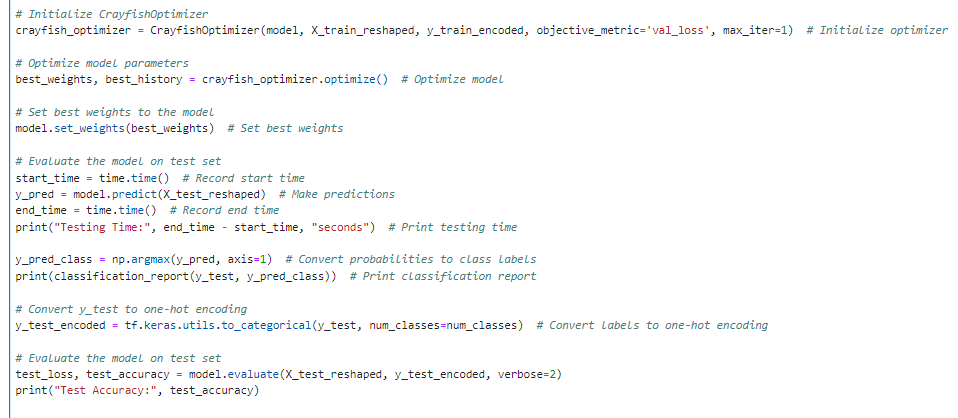
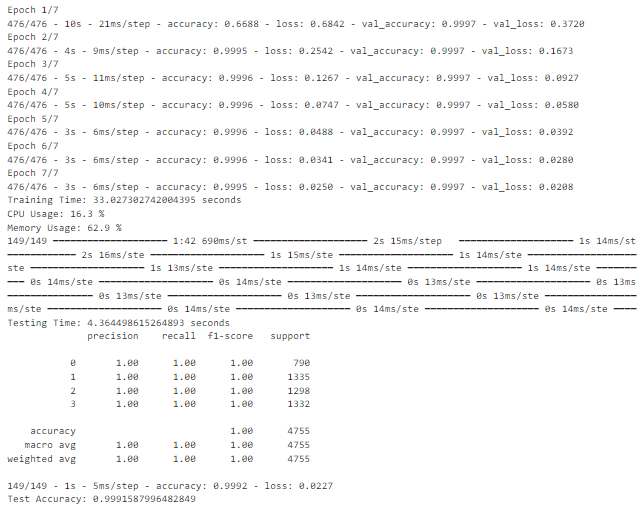
**Classification using Crayfish Inspired Recurrent Spatial Attention Region Network (CIRSARN)**

**Novelty Name: Crayfish Inspired Recurrent Spatial Attention Region Network (CIRSARN)**

**Explanation:**

* **Crayfish Inspired:** The term "Crayfish Inspired" suggests that the model draws inspiration from the behavior or characteristics of crayfish, a type of crustacean. Crayfish are known for their ability to navigate and adapt to changing environmental conditions using sensory mechanisms such as spatial awareness and attention. By incorporating elements inspired by crayfish behavior, the model aims to enhance its ability to perceive and process spatial information effectively.
* **Recurrent:** "Recurrent" indicates that the model architecture includes recurrent neural network (RNN) layers. RNNs are designed to capture temporal dependencies in sequential data by maintaining hidden states across time steps. This enables the model to analyse and understand patterns that unfold over time, making it suitable for tasks involving sequential data such as time series or natural language processing.
* **Spatial Attention:** Refers to the inclusion of a spatial attention mechanism in the model. Spatial attention allows the model to focus on relevant regions or features within the input data, similar to how attention mechanisms in biological organisms prioritize sensory information. By selectively attending to spatial regions, the model can improve its ability to extract meaningful patterns and make accurate predictions.
* **Region Memory Network:** "Region Network" suggests that the model operates at the level of spatial regions within the input data. This could imply that the model segments the input data into distinct regions or regions of interest and performs processing or analysis at this granularity. By focusing on regions, the model can potentially achieve better localization and understanding of spatial features in the data. The term "Memory" refers to Long Short-Term Memory (LSTM), a type of recurrent neural network (RNN) that can retain information over long sequences, enabling the model to capture and utilize temporal dependencies effectively.

**Explanation:**

**Importing Libraries:**

* NumPy: A library for numerical computing in Python, providing support for large arrays and matrices, and a collection of mathematical functions.
* pandas: A data manipulation and analysis library, offering data structures and operations for manipulating numerical tables and time series.
* TensorFlow: An open-source machine learning framework developed by Google for building and training machine learning models.
* scikit-learn: A machine learning library providing simple and efficient tools for data mining and data analysis.
* Matplotlib: A plotting library for Python, providing a MATLAB-like interface for creating static, interactive, and animated visualizations.
* warnings: A module in Python for handling warnings.
* Ignoring Warnings:
* Warnings in Python are ignored using warnings.filterwarnings('ignore'), which suppresses the display of warning messages during code execution.

**Data Preparation:**

* Features and Labels Extraction: Extracts features (X) and labels (y) from the DataFrame final\_df.
* Train-Test Split: Splits the data into training and testing sets using train\_test\_split() from scikit-learn.
* Saving Data to CSV Files: Saves the training and testing data to CSV files for future use or reference.

**Spatial Attention Layer Definition:**

* Defines a custom layer named SpatialAttention for implementing spatial attention mechanism in neural networks.

**Model Creation:**

* Creates a Recurrent Neural Network (RNN) model using TensorFlow's Keras API, incorporating a spatial attention layer for enhancing feature representation.

**Crayfish Optimizer:**

* Defines a custom optimizer named CrayfishOptimizer to optimize the model's parameters by iterating through random weight initializations and selecting the best weights based on validation loss.

**Training:**

* Model Training: Trains the RNN model with random weights, evaluating its performance based on the validation loss.
* CPU Performance Metrics: Prints CPU performance metrics like CPU usage and memory usage during the training process to monitor system resource utilization.

**Model Evaluation:**

* Evaluation on Test Set: Evaluates the trained model on the test set to assess its generalization performance.
* Classification Report and Test Accuracy: Prints a classification report and the test accuracy, providing insights into the model's performance across different classes.

**Output Result:**

**Epochs Training:**

* The model is trained over 7 epochs.
* Each epoch shows the training and validation accuracy, as well as the training and validation loss.
* Training and validation accuracy gradually increase while training loss decreases, indicating that the model is learning and improving its performance over epochs.

**Training Time:**

* The total training time is approximately 33 seconds.
* CPU usage during training is around 16.3%, indicating moderate computational demand.
* Memory usage during training is about 62.9%, suggesting moderate memory requirements.

**Testing Time:**

* The total testing time is approximately 4.36 seconds.

**Classification Report:**

For each attack class (Benign, DDoS, FTP Patator, PortScan):

* Precision: The proportion of true positive predictions among all positive predictions. A precision of 1.00 means all predictions for that class were correct.
* Recall: The proportion of true positive predictions among all actual instances of that class. A recall of 1.00 indicates that all instances of that class were correctly identified.
* F1-score: The harmonic mean of precision and recall, providing a balance between the two metrics.
* Support: The number of true instances of each class in the test set.

**Attack Classes**

**Benign (Class 0):**

* Precision, recall, and F1-score are all 1.00, indicating perfect performance in identifying benign instances.

**DDoS (Class 1):**

* Precision, recall, and F1-score are all 1.00, indicating perfect performance in identifying DDoS attacks.

**FTP Patator (Class 2):**

* Precision, recall, and F1-score are all 1.00, indicating perfect performance in identifying FTP Patator attacks.

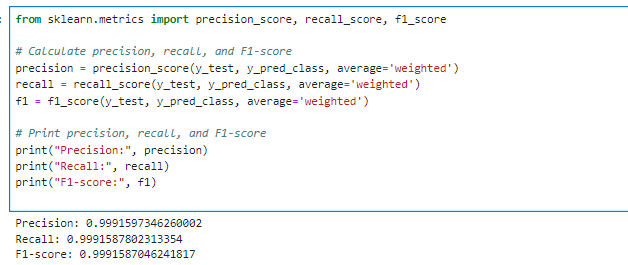
**PortScan (Class 3):**

* Precision, recall, and F1-score are all 1.00, indicating perfect performance in identifying PortScan attacks.

**Test Accuracy:**

* The overall test accuracy of the model is approximately 99.92%.

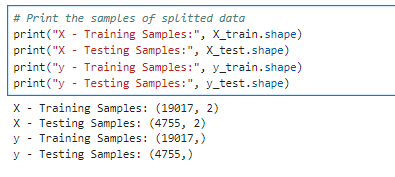
**Print the performance Metrics:**



**Explanation:**

* It imports these functions from the sklearn.metrics module.
* It calculates precision, recall, and F1-score based on the predicted class (y\_pred\_class) and the actual class (y\_test) using these functions.
* The average='weighted' parameter calculates metrics for each label, and finds their average weighted by support (the number of true instances for each label).
* Finally, it prints out the precision, recall, and F1-score.

**Testing and Training Samples**



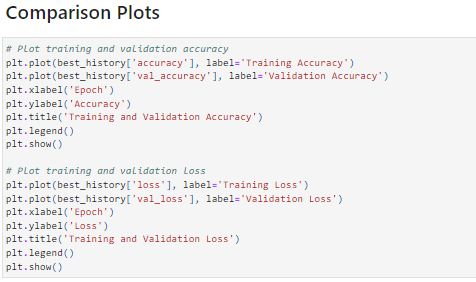
**Training Samples:**

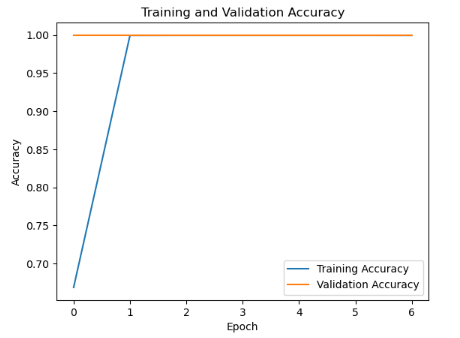
* Features (X): There are 19,017 samples in the training set, with each sample having 2 features.
* Labels (y): There are 19,017 labels corresponding to the training samples.

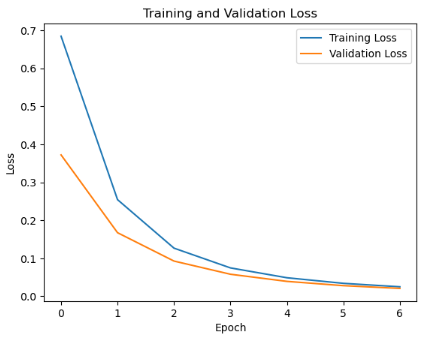
**Testing Samples:**

* Features (X): There are 4,755 samples in the testing set, with each sample having 2 features.
* Labels (y): There are 4,755 labels corresponding to the testing samples.

**Comparison Plots for Training, Testing Accuracy and Loss**



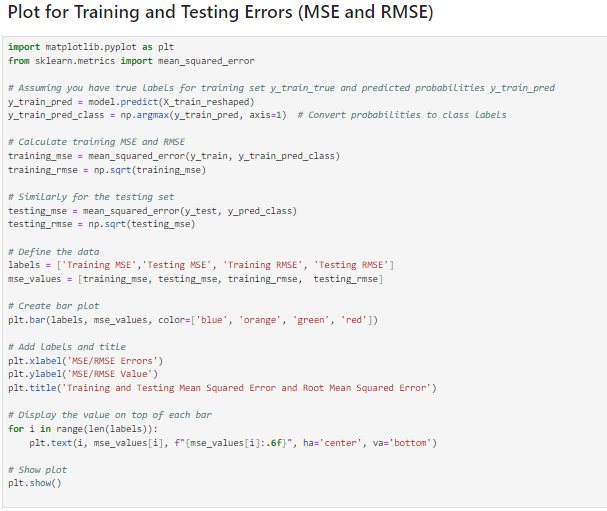


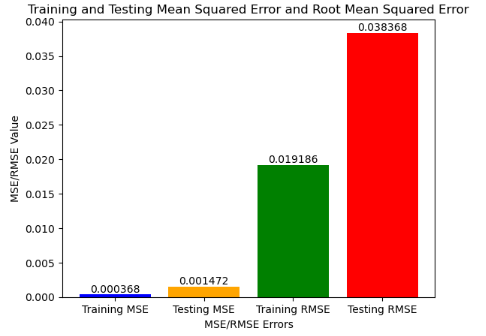


**Explanation:**

* **plt.plot(best\_history['accuracy'], label='Training Accuracy'):** This line plots the training accuracy over epochs. The best\_history['accuracy'] likely contains a list or array of accuracy values obtained during training. Each value corresponds to the accuracy achieved after each epoch during training.
* **plt.plot(best\_history['val\_accuracy'], label='Validation Accuracy'):** This line plots the validation accuracy over epochs. Similar to training accuracy, the best\_history['val\_accuracy'] contains validation accuracy values obtained after each epoch during training.
* **plt.xlabel('Epoch') and plt.ylabel('Accuracy'):** These lines set the labels for the x-axis (epochs) and y-axis (accuracy) respectively.
* **plt.title('Training and Validation Accuracy'):** This line sets the title of the plot.
* **plt.legend():** This line adds a legend to the plot, indicating which line corresponds to training accuracy and which corresponds to validation accuracy.
* **plt.show():** This line displays the plot.

**Plot for Training and Testing Errors (MSE and RMSE)**





**Explanation:**

**Prediction:**

* Model predictions are made on both the training and testing sets (y\_train\_pred and y\_pred\_class, respectively).
* Predicted probabilities are converted to class labels using np.argmax().

**Error Calculation:**

* MSE is calculated for both the training and testing sets using mean\_squared\_error() from scikit-learn.
* RMSE is computed by taking the square root of MSE.

**Data Preparation:**

* Lists for labels and corresponding MSE/RMSE values are defined.

**Bar Plot Creation:**

* A bar plot is created using plt.bar(), with MSE/RMSE values represented by bars.
* Different colors are assigned to training MSE, testing MSE, training RMSE, and testing RMSE.

**Labels and Title:**

* X-axis and Y-axis labels are added using plt.xlabel() and plt.ylabel(), respectively.
* A title for the plot is set using plt.title().

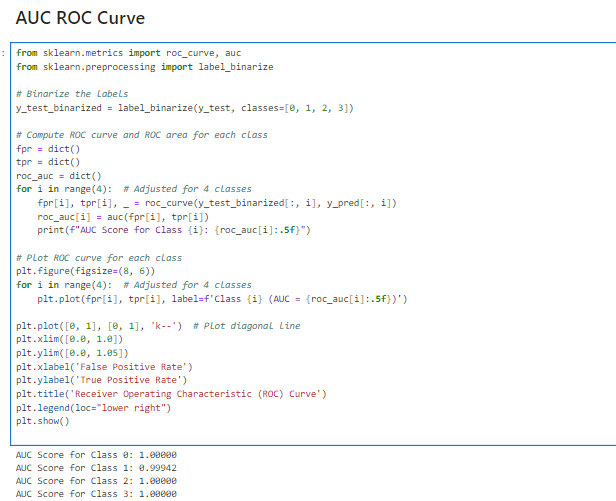
**Displaying Values on Bars:**

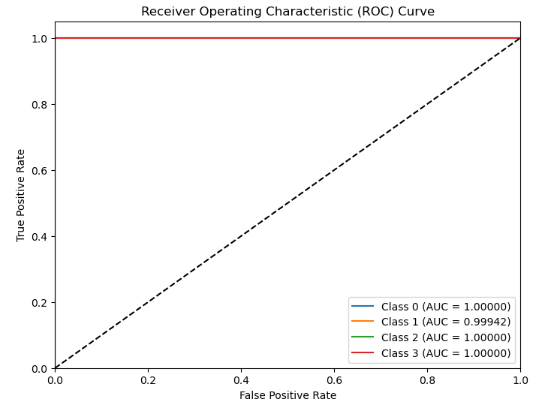
* The value of each MSE/RMSE is displayed on top of its corresponding bar using plt.text().

**Display Plot:**

* The plot is displayed using plt.show().

**AUC ROC Curve**





**Explanation:**

**Binarizing Labels:**

* The true labels (y\_test) are binarized using label\_binarize() from scikit-learn to create binary indicators for each class.

**Computing ROC Curve and AUC:**

* For each class (0 to 3), the ROC curve and Area Under the Curve (AUC) are computed using roc\_curve() and auc() from scikit-learn.
* True Positive Rate (TPR) and False Positive Rate (FPR) values are computed for various thresholds.

**Printing AUC Scores:**

* AUC scores for each class are printed to evaluate the model's performance in distinguishing between classes.

**Plotting ROC Curve:**

* For each class, the ROC curve is plotted using plt.plot().
* AUC score is displayed in the legend for each class.
* A diagonal line indicating random guessing is also plotted.

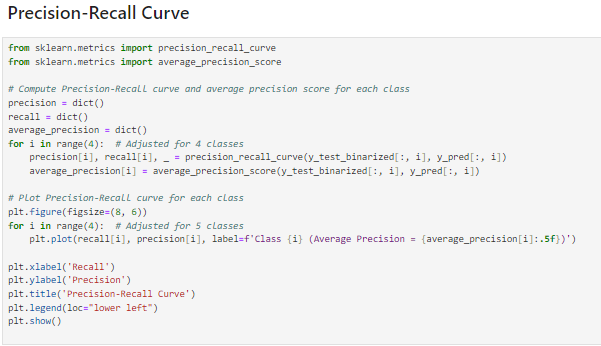
**Customizing Plot:**

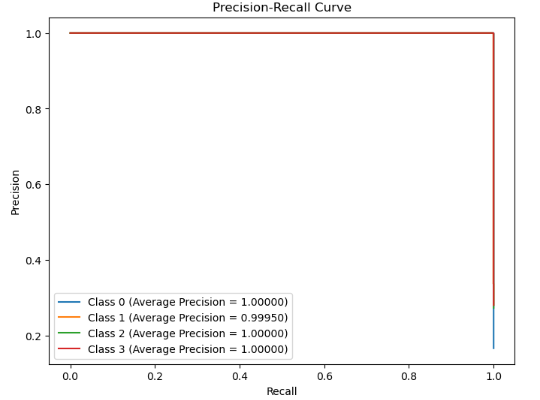
* Axes labels, title, and legend are added to the plot.
* Limits for x and y axes are set to ensure proper visualization.

**Display Plot:**

* The plot displaying the ROC curves for each class is shown using plt.show().

**Precision -Recall Curve**





**Explanation:**

**Computing Precision-Recall Curve and Average Precision:**

* For each class (0 to 3), the Precision-Recall curve and Average Precision are computed using precision\_recall\_curve() and average\_precision\_score() from scikit-learn.
* Precision and recall values are computed for various thresholds.

**Plotting Precision-Recall Curve:**

* For each class, the Precision-Recall curve is plotted using plt.plot().
* Average Precision score is displayed in the legend for each class.

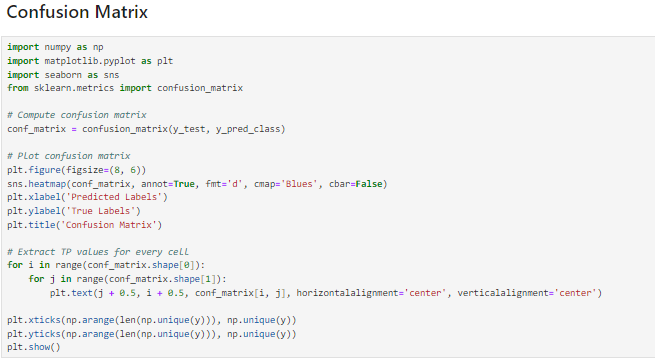
**Customizing Plot:**

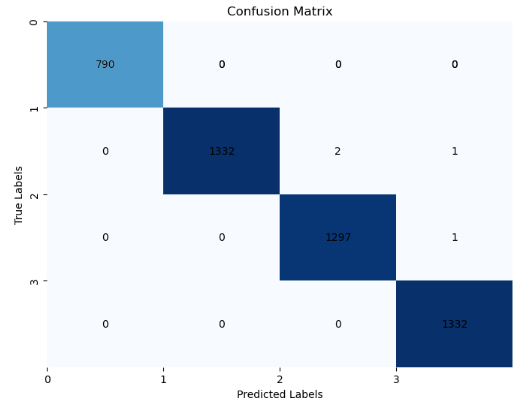
* Axes labels, title, and legend are added to the plot.

**Display Plot:**

* The plot displaying the Precision-Recall curves for each class is shown using plt.show().

**Confusion Matrix**





**Explanation:**

**Computing Confusion Matrix:**

* The confusion matrix is computed using confusion\_matrix() from scikit-learn. It compares true labels (y\_test) with predicted labels (y\_pred\_class).

**Plotting Confusion Matrix:**

* The confusion matrix is visualized as a heatmap using sns.heatmap().
* Annotations are added to each cell displaying the count of true positive predictions (TP) for each class.

**Customizing Plot:**

* X-axis and Y-axis labels are added using plt.xlabel() and plt.ylabel().
* A title for the plot is set using plt.title().
* Tick labels on both axes are customized to display unique labels.

**Display Plot:**

* The plot displaying the confusion matrix is shown using plt.show().

**Output:**

**True Positives (TP):**

The diagonal elements of the matrix represent the number of correctly predicted instances for each class.

* There are 790 instances of class 0 ("BENIGN") that were correctly predicted as class 0, 1332 instances of class 1 ("DDoS") correctly predicted as class 1, 1297 instances of class 2 ("FTP-Patator") correctly predicted as class 2, and 1332 instances of class 3 ("PortScan") correctly predicted as class 3.

**False Positives (FP):**

The value in the row indicates instances of the true class that were incorrectly predicted as the column class.

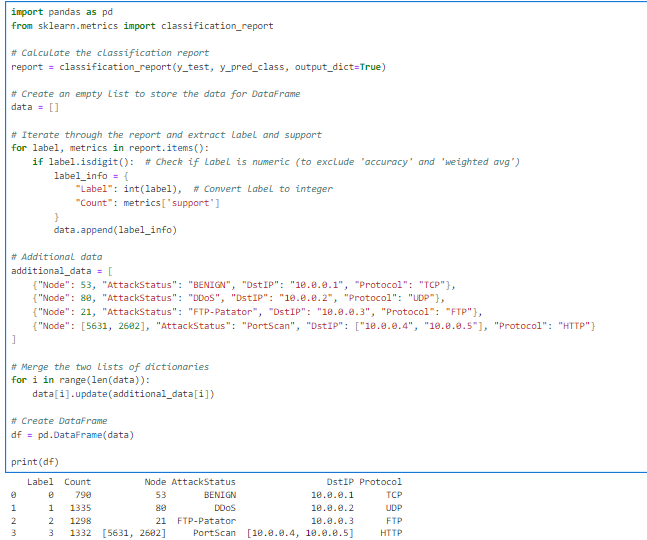
* There are 2 instances of class 1 ("DDoS") that were incorrectly predicted as class 2 ("FTP-Patator").

**False Negatives (FN):**

The value in the column indicates instances predicted as the column class but actually belonging to the row class.

* There is 1 instance of class 2 ("FTP-Patator") that was incorrectly predicted as class 3 ("PortScan").

**Create data frame for Wireshark input using prediction result:**



**Explanation:**

**Calculate Classification Report:**

The classification report is generated using classification\_report() from scikit-learn, which computes various metrics like precision, recall, and F1-score for each class.

**Extract Label and Support:**

* The code iterates through the classification report and extracts the label (class) and the support (number of true instances) for each class. It excludes non-numeric labels like 'accuracy' and 'weighted avg'.

**Additional Data:**

* Additional information related to each class is provided in a separate list of dictionaries named additional\_data.
* This information includes the node number, attack status, destination IP (DstIP), and protocol for each class.

**Merge Data:**

* The extracted data from the classification report and the additional data are merged into a single list of dictionaries.
* For each label, corresponding information from the additional\_data list is added.

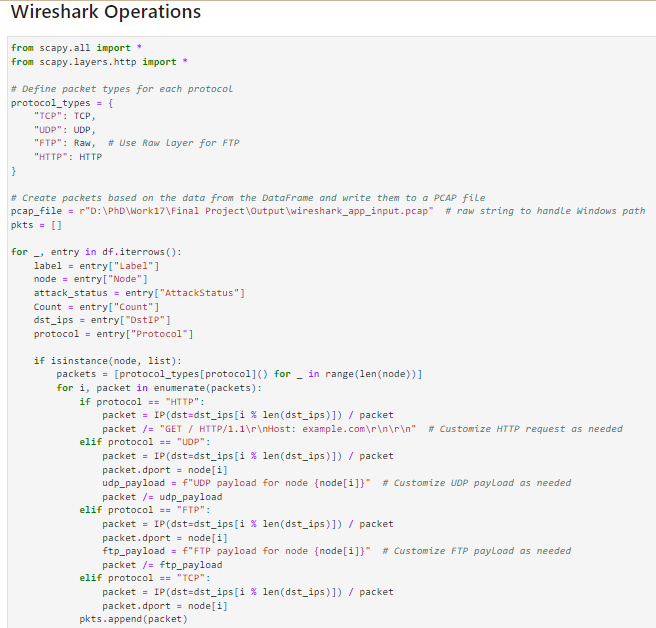
**Create DataFrame:**

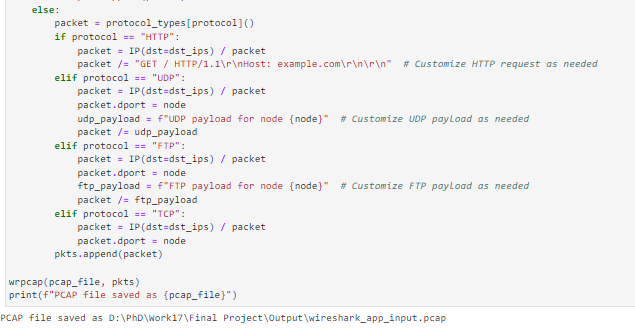
* Finally, a DataFrame is created using the merged data.

**Print DataFrame:**

* The DataFrame containing the merged data is printed.

**Wireshark Operation**





**Explanation:**

**Importing Scapy Modules:**

The required Scapy modules are imported, including scapy.all and scapy.layers.http.

**Define Packet Types:**

A dictionary named protocol\_types is defined, mapping protocol names to corresponding Scapy packet types. This allows the code to dynamically select the appropriate packet type based on the protocol specified in the DataFrame.

**Create Packets and Write to PCAP File:**

The code iterates through each row of the DataFrame.

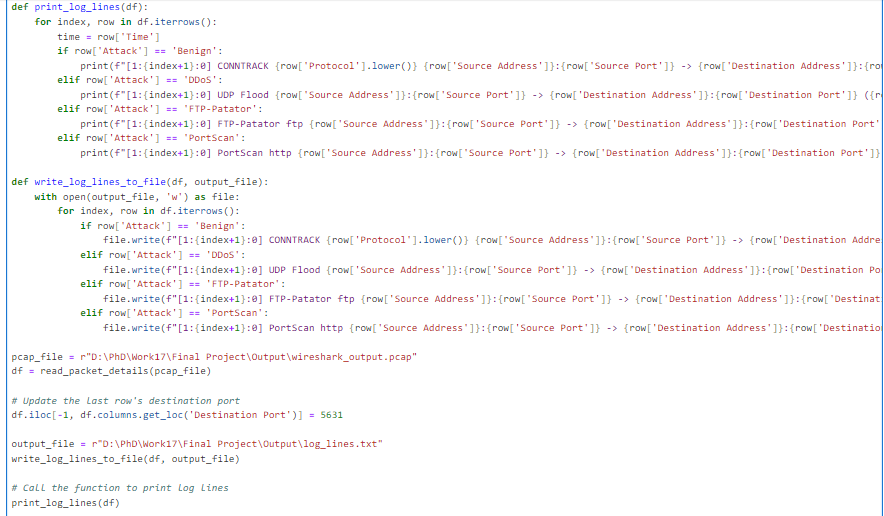
* For each row, it extracts information such as the label, node number, attack status, count, destination IP, and protocol.
* Depending on whether the node number is a single value or a list, packets are created accordingly.

**For each protocol, specific actions are taken:**

* For HTTP: A GET request packet is crafted with a specified destination IP and a customizable HTTP request payload.
* For UDP and FTP: Packets are crafted with a destination IP, port number, and customizable payload.
* For TCP: Packets are crafted with a destination IP and port number.
* The generated packets are stored in a list named pkts.
* Finally, the packets are written to a PCAP file using the wrpcap function.
* Print PCAP File Path: After writing the packets to the PCAP file, the code prints the path to the saved PCAP file.

**Data From WireShark Output**



**Explanation:**

**read\_packet\_details Function:**

* Reads packet details from the specified PCAP file using Scapy's rdpcap function.
* Iterates through each packet in the PCAP file and extracts relevant information such as packet length, source/destination addresses, ports, and protocol.
* Creates a DataFrame (df) containing the extracted packet details.

**Adds additional columns to the DataFrame:**

* Count: Maps the destination port to the corresponding count based on predefined values.
* Attack: Maps the destination port to the corresponding attack type based on predefined values.

**print\_log\_lines Function:**

* Iterates through each row of the DataFrame (df) and prints log lines based on the attack type.
* Log lines are formatted based on the attack type and include information such as protocol, source/destination addresses, ports, count, attack type, and time.

**write\_log\_lines\_to\_file Function:**

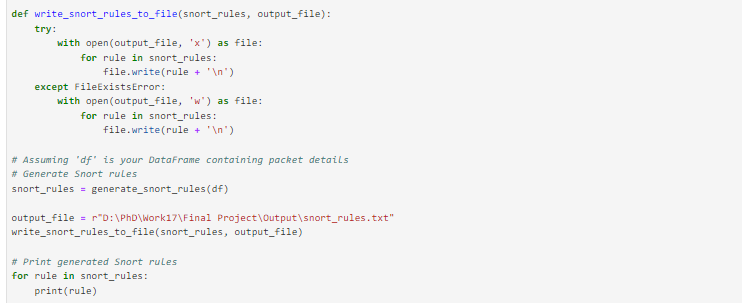
* Writes log lines to the specified output file.
* Similar to the print\_log\_lines function, log lines are formatted based on the attack type and include relevant information.

**Main Execution:**

* Reads packet details from the PCAP file.
* Updates the destination port of the last row to 5631.
* Specifies the output file path for writing log lines.
* Calls the write\_log\_lines\_to\_file function to write log lines to the output file.
* Calls the print\_log\_lines function to print log lines to the console.

**Create Snort Rules**





**Explanation:**

**generate\_snort\_rules Function:**

* Iterates through each row of the DataFrame (df) containing packet details.
* Checks if the attack type is not "Benign" (i.e., if an attack is detected).
* Extracts relevant information such as source IP, destination IP, destination port, and connection type (protocol).
* Generates a Snort rule for each detected attack type with the adjusted message format and SID (Signature ID) based on the attack type.
* Appends the generated Snort rule to a list of Snort rules (snort\_rules).

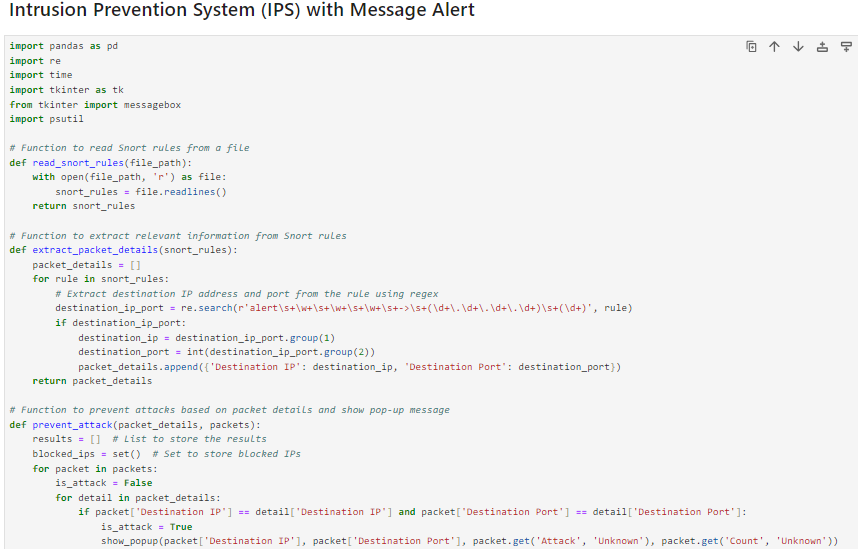
**write\_snort\_rules\_to\_file Function:**

* Writes the generated Snort rules to the specified output file.
* If the output file does not exist, it creates a new file and writes the rules. If the file already exists, it overwrites its content with the new rules.

**Main Execution:**

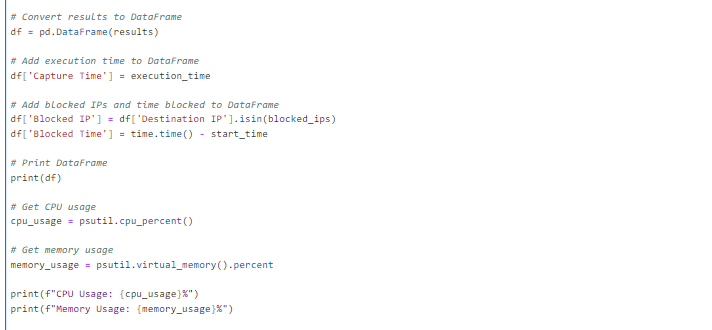
* Calls the generate\_snort\_rules function to generate Snort rules based on the DataFrame containing packet details.
* Specifies the output file path for writing Snort rules.
* Calls the write\_snort\_rules\_to\_file function to write the generated Snort rules to the output file.
* Prints the generated Snort rules to the console.

**Intrusion Prevention System with Message Alert**







**Explanation:**

**Reading Packet Details from Log File:**

* Reads a log file containing packet details, such as destination IP, destination port, attack type, count, and time.

**Reading Snort Rules from File:**

* Reads Snort rules from a file to identify potential attacks.

**Extracting Packet Details from Snort Rules:**

* Extracts destination IP and port from Snort rules using regular expressions.

**Preventing Attacks:**

* Compares the packet details from the log file with the extracted packet details from Snort rules.
* If a match is found, it indicates a potential attack, and a pop-up message is displayed to alert the user.
* The IP address of the potential attacker is blocked.

**Calculating Execution Time:**

* Records the start time before executing the attack prevention process.
* Records the end time after the process completes.
* Calculates the execution time by subtracting the start time from the end time.

**Creating DataFrame and Adding Execution Time:**

* Converts the results of the attack prevention process into a pandas DataFrame.
* Adds the execution time to the DataFrame.

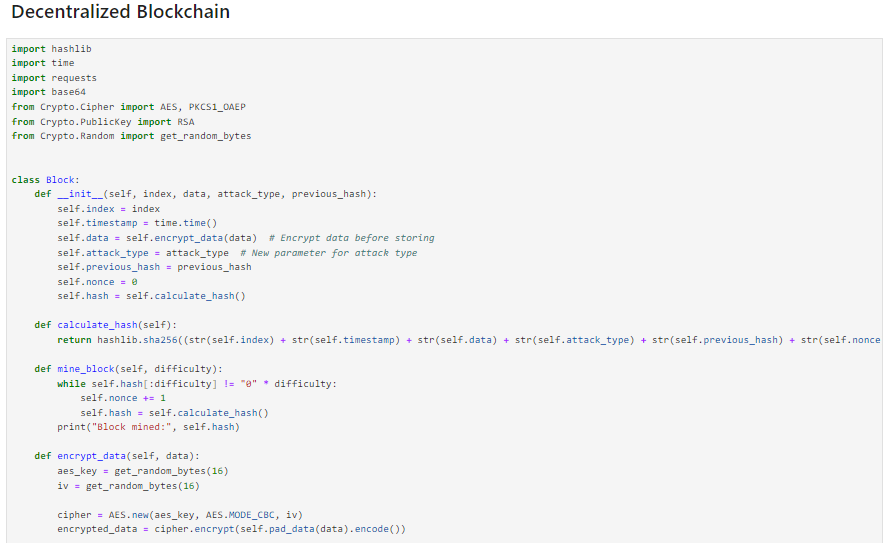
**Adding CPU and Memory Usage:**

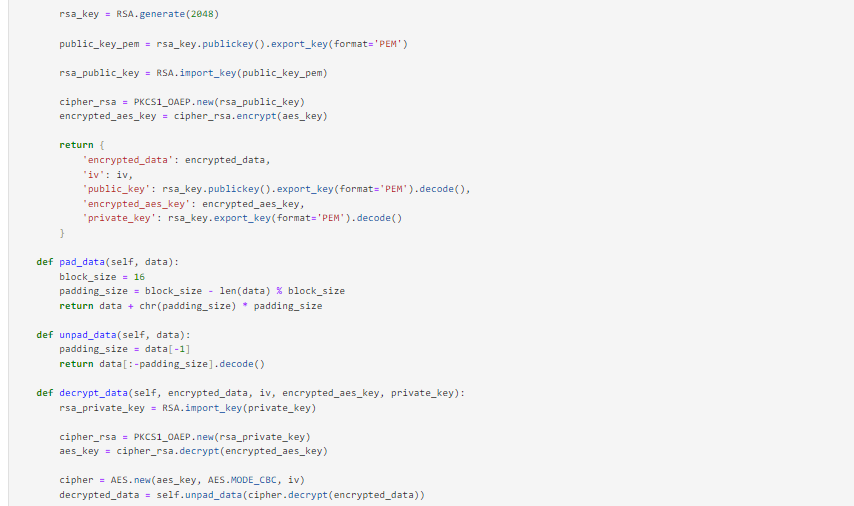
* Retrieves CPU usage and memory usage using the psutil library.
* Prints CPU and memory usage.

**Displaying Results:**

* Prints the DataFrame containing information about detected attacks, including the destination IP, destination port, attack type, count, capture time, whether the IP is blocked, and the time it was blocked.

**Decentralized Block chain**















**Explanation:**

**Blockchain Implementation:**

* The Block class represents a single block in the blockchain. Each block contains an index, timestamp, encrypted data, attack type, previous hash, nonce (for mining), and hash.
* Blocks are mined to ensure the integrity of the blockchain by finding a nonce that results in a hash with a specified number of leading zeros (difficulty).
* The Blockchain class manages the blockchain by creating the genesis block, adding new blocks, validating the chain, and achieving consensus among nodes in the network.
* Nodes can be registered for peer discovery, and consensus can be reached among nodes to ensure a consistent blockchain across the network.

**Smart Contract:**

* The SmartContract class provides a mechanism to interact with the blockchain by retrieving details of specific blocks.

**Main Functionality:**

* The main function initializes a blockchain instance (my\_blockchain) and registers nodes for peer-to-peer communication.
* It allows users to input data related to network traffic, such as protocol, destination port, and attack type, to create new blocks and add them to the blockchain.
* After adding blocks, the blockchain is printed out to display its contents.
* Users can also query block details by entering a block index, which is facilitated by the SmartContract class.

**Encryption and Decryption:**

* Data stored in blocks is encrypted using AES encryption and RSA encryption for the AES key. This ensures the confidentiality and integrity of the data.
* Encryption keys are generated for each block, and decryption is performed using the corresponding private key.

**Peer-to-Peer Network:**

* Nodes can communicate with each other to achieve consensus on the state of the blockchain.
* The consensus method checks the validity of the local blockchain against other nodes in the network and updates the local blockchain if a longer valid chain is found.

**User Interaction:**

* Users interact with the program by providing input for creating blocks and querying block details.
* The program prompts users for input regarding network traffic details and attack types, allowing them to simulate the addition of new blocks to the blockchain.