**STREAM ANALYTICS**

**CSL7970**

**POST GRADUATE DIPLOMA**

**IN DATA ENGINEERING**

## ASSIGNMENT 1

**SUBMITTED BY:**

**NIRAJ BHAGCHANDANI [G23AI2087]**

**BHAVESH ARORA [G23AI2126]**

**A logo with a circular design

Description automatically generated with medium confidence**

**COURSE INSTRUCTOR**

**NITIN AWATHARE**

**SUBMISSION DATE: 23rd DECEMBER, 2024**

**DEPARTMENT OF AIDE**

**INDIAN INSTITUTE OF TECHNOLOGY, JODHPUR**

1. Retrieve the data stream that represents only the TCP traffic flowing through the network. To do this, you should become familiar with the SiLK suite, a tool for analyzing the network and its components.

Step 1: Installing various packages as well as downloading certain files from NETSA website, so here are the commands for the same.

$ sudo apt update

$ sudo apt upgrade -y

$ sudo apt install dos2unix -y

$ sudo apt install build-essential -y

$ sudo apt-get install s3cmd -y

$ sudo apt install libglib2.0-dev liblzo2-dev zlib1g-dev libgnutls28-dev libpcap-dev python3.8-dev

$ sudo apt install libmaxminddb-dev libssl-dev

$ sudo apt install libpcre3 libpcre3-dev

$ sudo apt install unzip

$ sudo apt install libssl-dev

$ sudo apt install build-essential autoconf automake

$ sudo apt install build-essential autoconf automake libtool libpcre3-dev libssl-dev

$ sudo apt install gcc-11 g++-11

$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

$ unzip awscliv2.zip

$ sudo ./aws/install

$ aws --version

$ wget https://tools.netsa.cert.org/releases/yaf-2.16.1.tar.gz

$ wget https://tools.netsa.cert.org/releases/silk-3.23.1.tar.gz

$ wget https://tools.netsa.cert.org/releases/libfixbuf-2.5.0.tar.gz

$ wget https://tools.netsa.cert.org/releases/analysis-pipeline-5.11.4.tar.gz

$ wget https://tools.netsa.cert.org/releases/netsa-python-1.5.tar.gz

$ wget https://tools.netsa.cert.org/releases/rayon-1.4.3.tar.gz

$ wget https://tools.netsa.cert.org/releases/libschemaTools-1.4.tar.gz

Step – 2 Installing pre-requisite softwares before installing packages.

A screenshot of a computer program

Description automatically generated

Step – 3 Downloading FCCX-pcap.tar.gz file from NETSA

A screenshot of a computer

Description automatically generated

Step – 4 Downloading FCCX-silk.tar.gz file from NETSA

A screenshot of a computer

Description automatically generated

Step-5 Configuring libfixbuf package

A screenshot of a computer

Description automatically generated

Step – 6 Compiling libfixbuf package

A screenshot of a computer program

Description automatically generated

Step – 7 Installing libfixbuf using make install command.

A screen shot of a computer

Description automatically generated

Step 8 Installing relevant packages for Silk Suite Support

A screenshot of a computer

Description automatically generated

Step – 8 Checking / Verifying the libfixbuf package installed correctly or not.

A computer screen shot of a program

Description automatically generated

Step 9 Coping certain files to ldconfig and reloading it.

A computer screen shot of a black screen

Description automatically generated

Step – 10 Extracting Silk-Suite using tar command.

A screenshot of a computer

Description automatically generated

Step – 11 Compiling Silk Suite using configure command.

A screen shot of a computer program

Description automatically generated

Step – 12 Builing Suilk Suite using make command.

A computer screen shot of a program

Description automatically generated

Step – 13 Installing Silk Suite using make install command

A screenshot of a computer program

Description automatically generated

Step – 14 Checking if rwflowpack is correctly installed or not.

A screenshot of a computer program

Description automatically generated

Step – 15 Exporting binary files to PATH environment variable for running the commands anywhere from the terminal.

A screenshot of a computer program

Description automatically generated

Step – 16 Checking where the file libflowsource files are located and thus, setting environment variable as such.

A screenshot of a computer screen

Description automatically generated

Step – 17 Checking the ouput of rwflowpack as well as setting up the environment variable also setting up the SILK\_CONFIG\_FILE for silk.conf file.

A computer screen shot of a program

Description automatically generated

Step – 18 Untar yaf file to home location of Ubuntu.

A screenshot of a computer

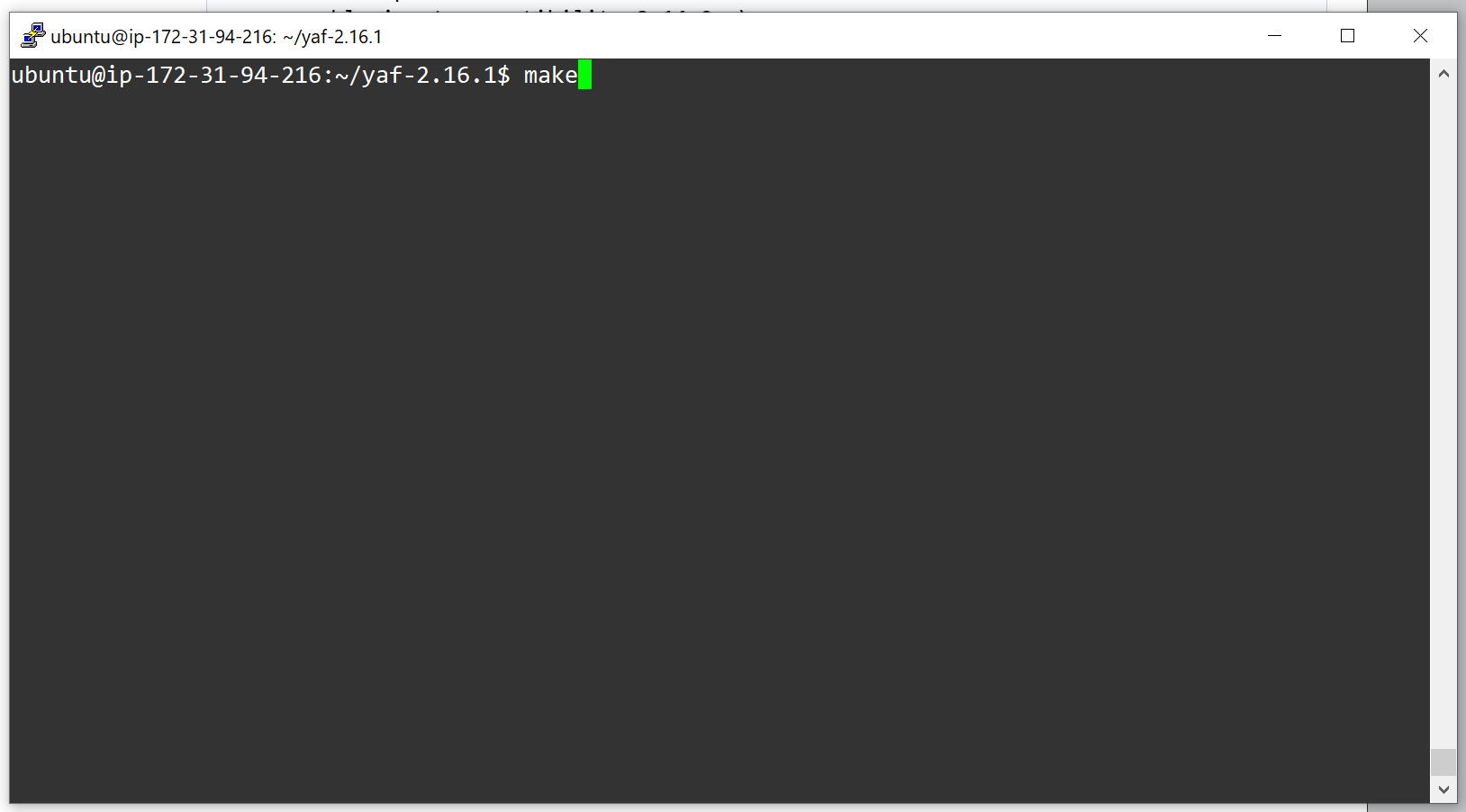
Description automatically generated

Step – 19 Compiling the yaf package using configure command.

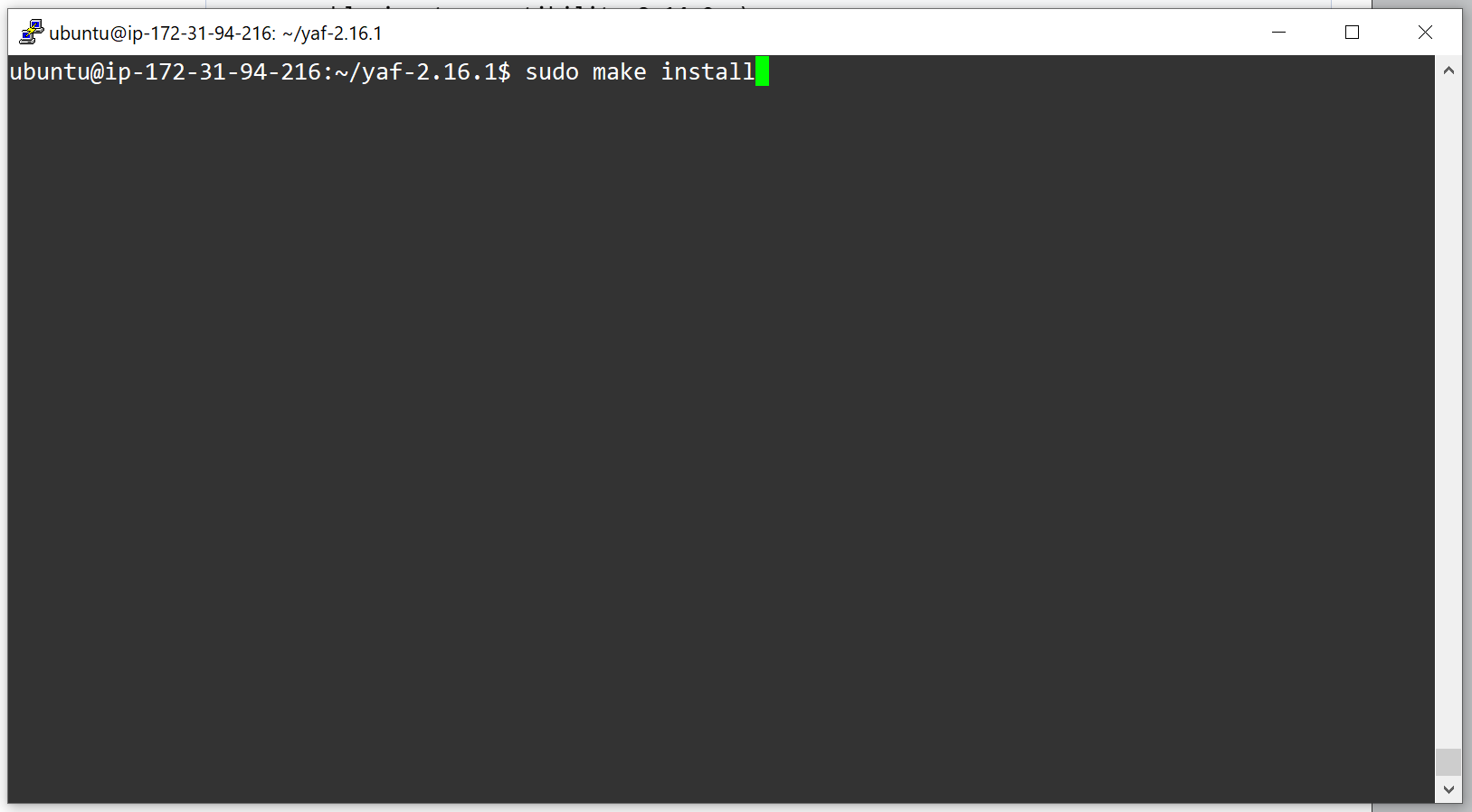
A screenshot of a computer

Description automatically generated

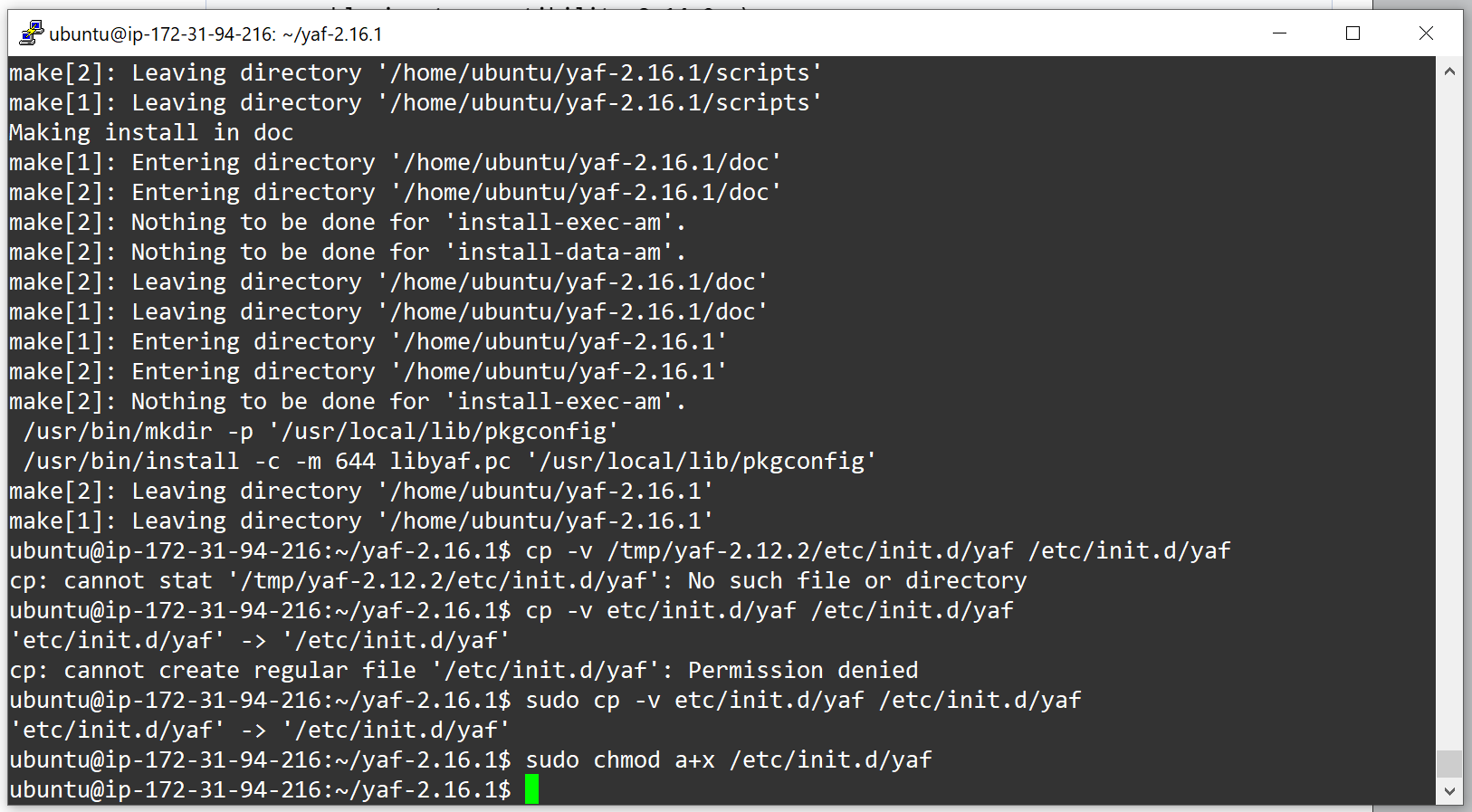
Step – 20 Building up the yaf package using make command.



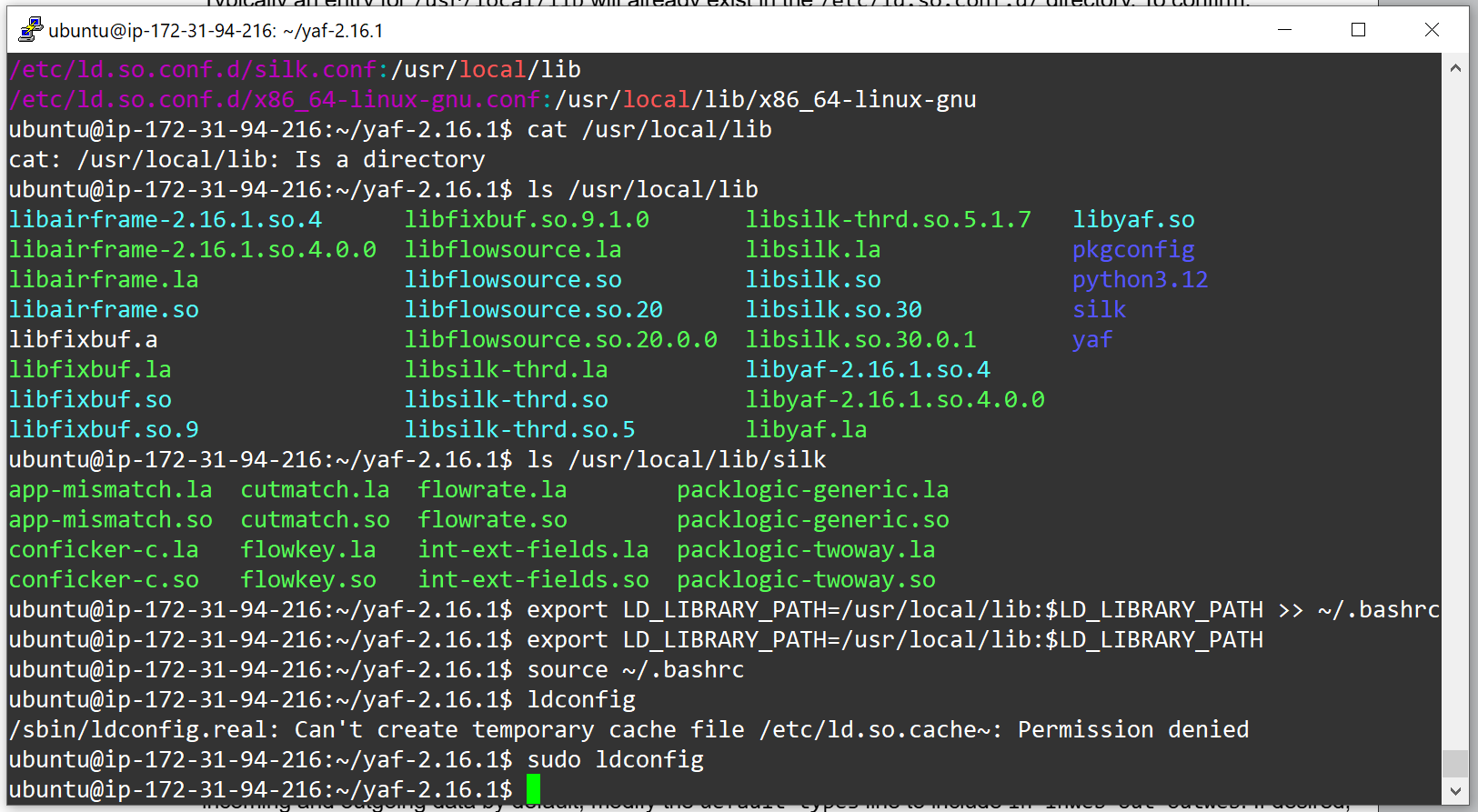
Step – 21 Instaling yaf package using make install command.



Step – 22 Setting up yaf package to autostart init.d /etc.init.d/ folder.



Step – 23 Adding environment Variable such as LD\_LIBRARY\_PATH also reconfiguring ldconfig command using ldconfig command.

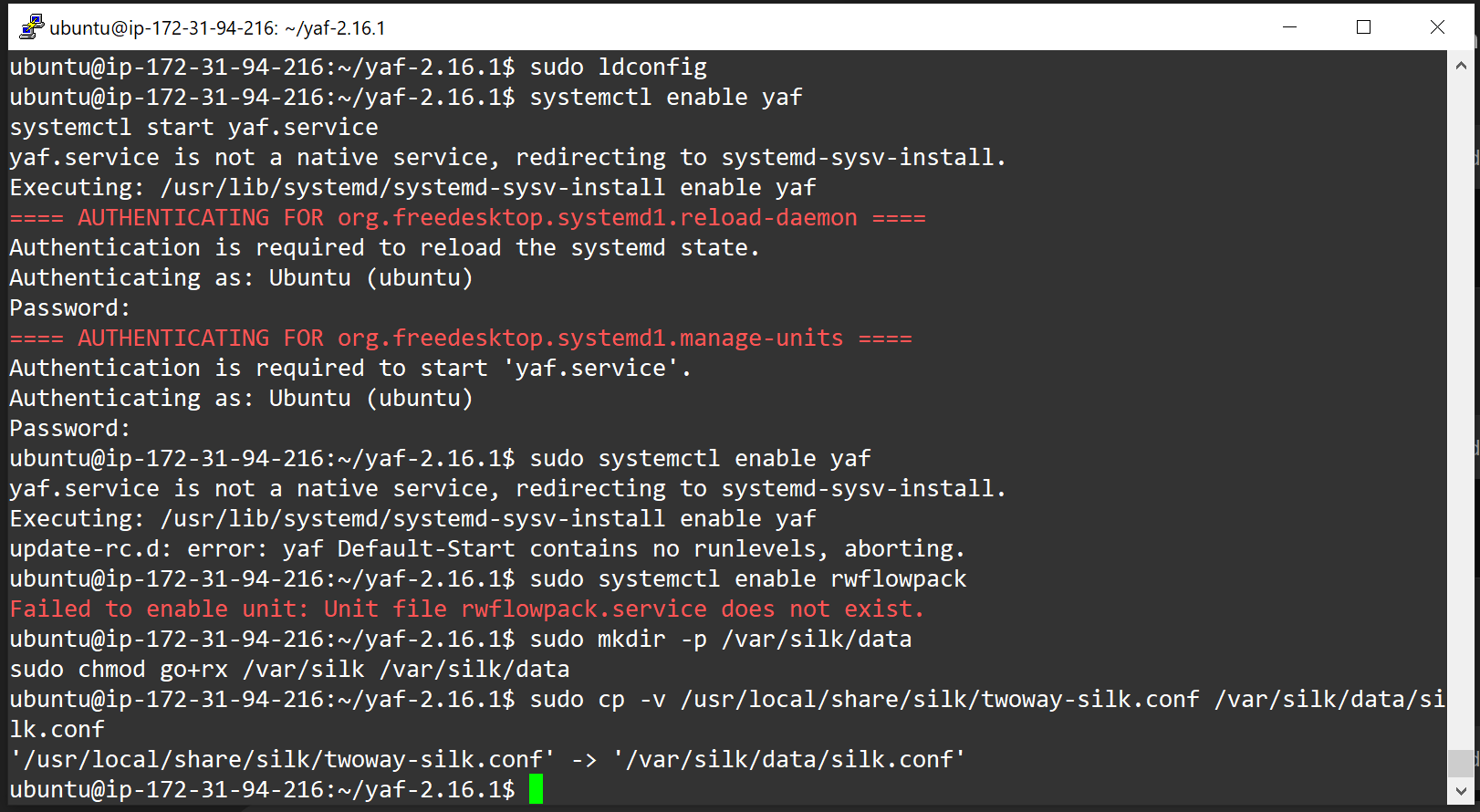


Step-24 Unizipping FCCX-silk.tar.gz using unzip command

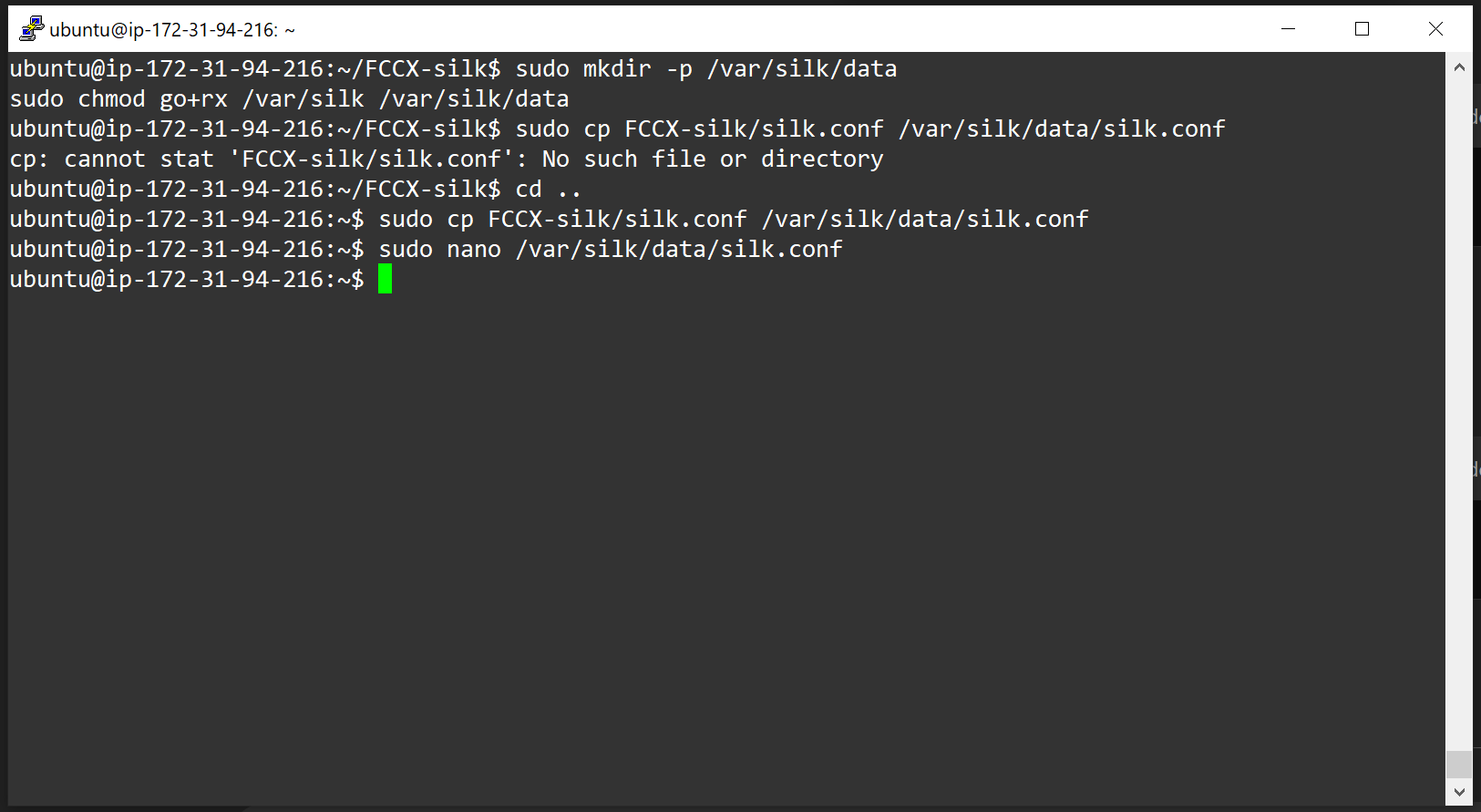
A screenshot of a computer program

Description automatically generated

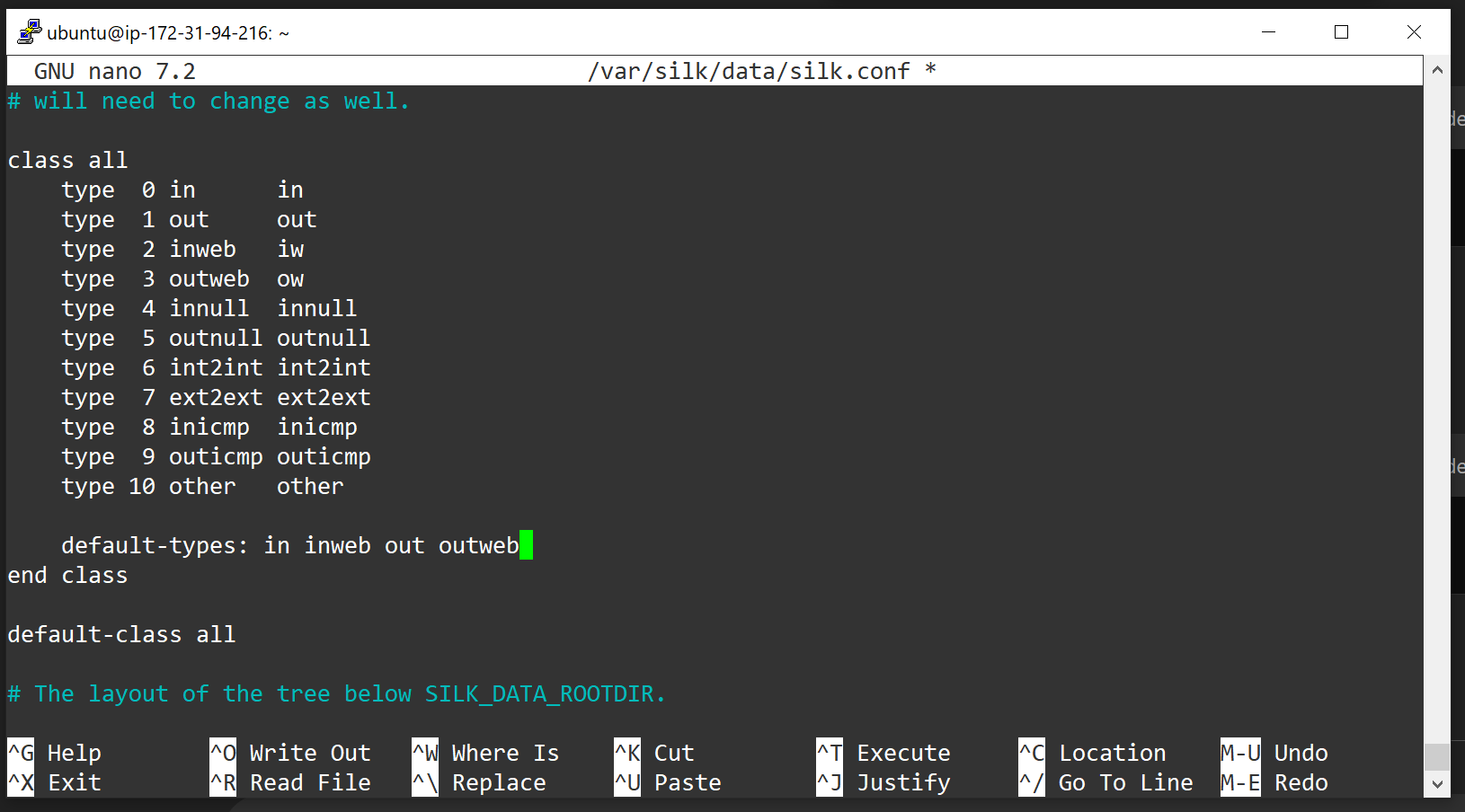
Step – 25 Enabling yaf to autostart also, copying twoway-silk.conf to its desired location.



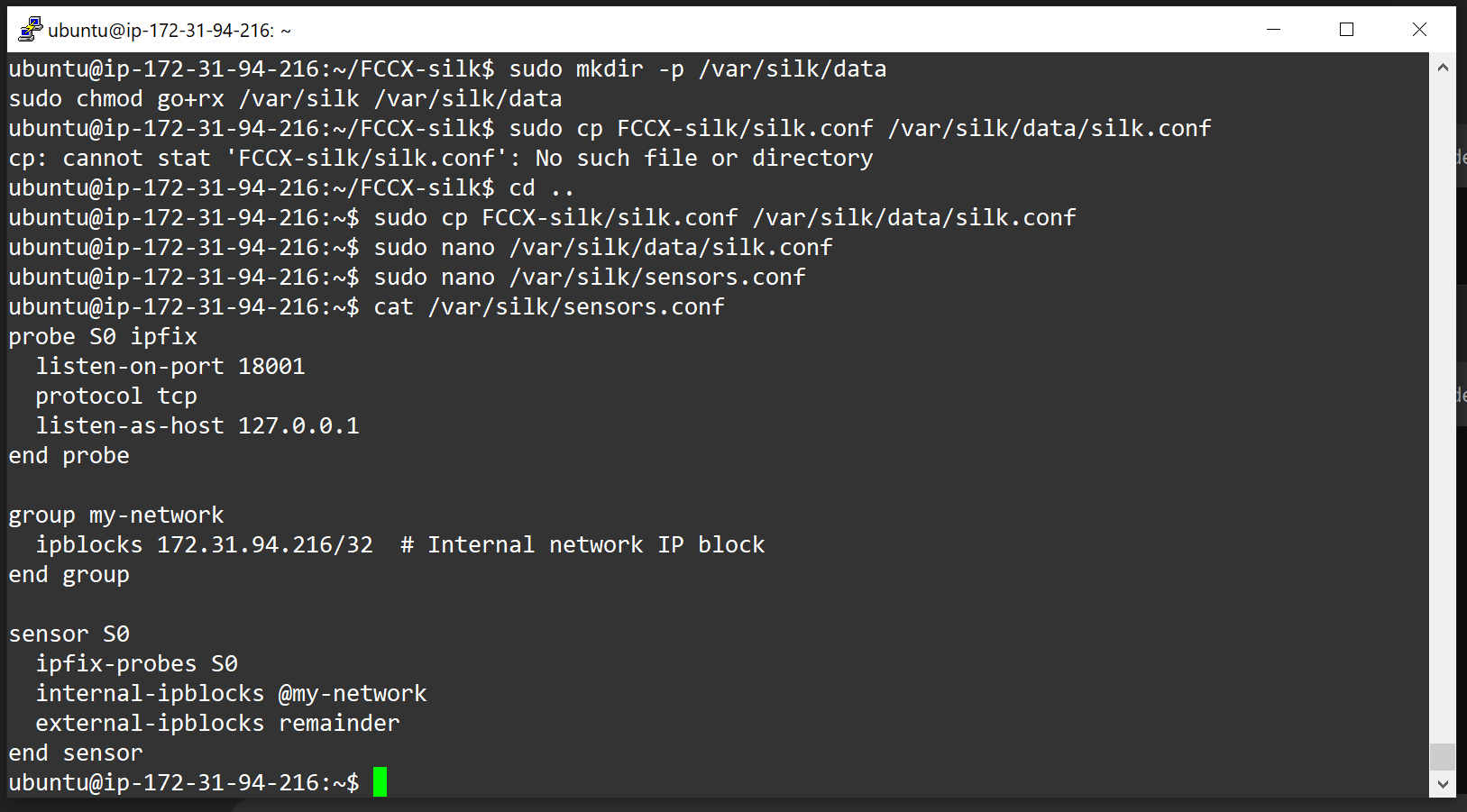
Step – 26 Editing certain parts of silk.conf file.



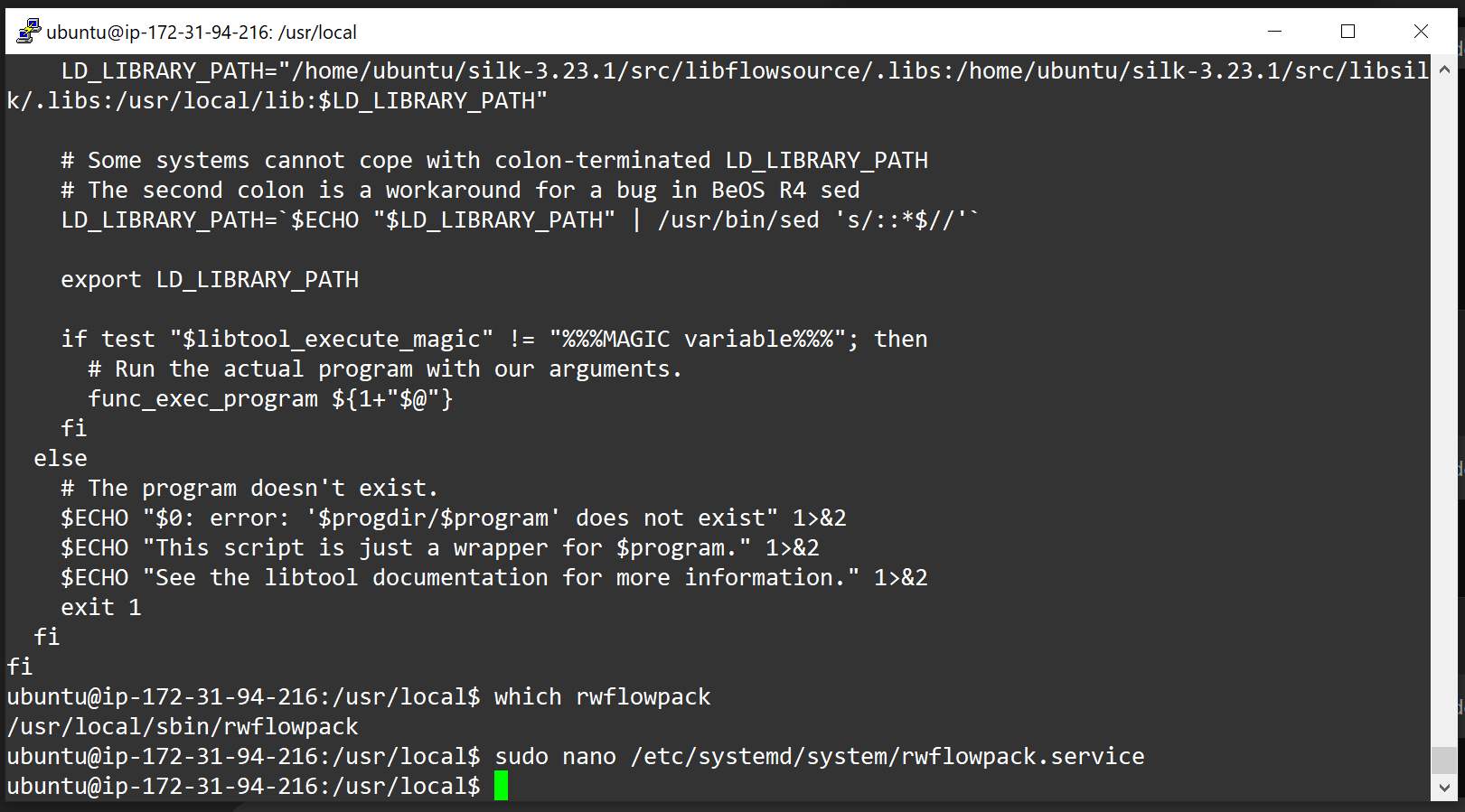
Step – 27 After Changing silk.conf file it should look something like this.



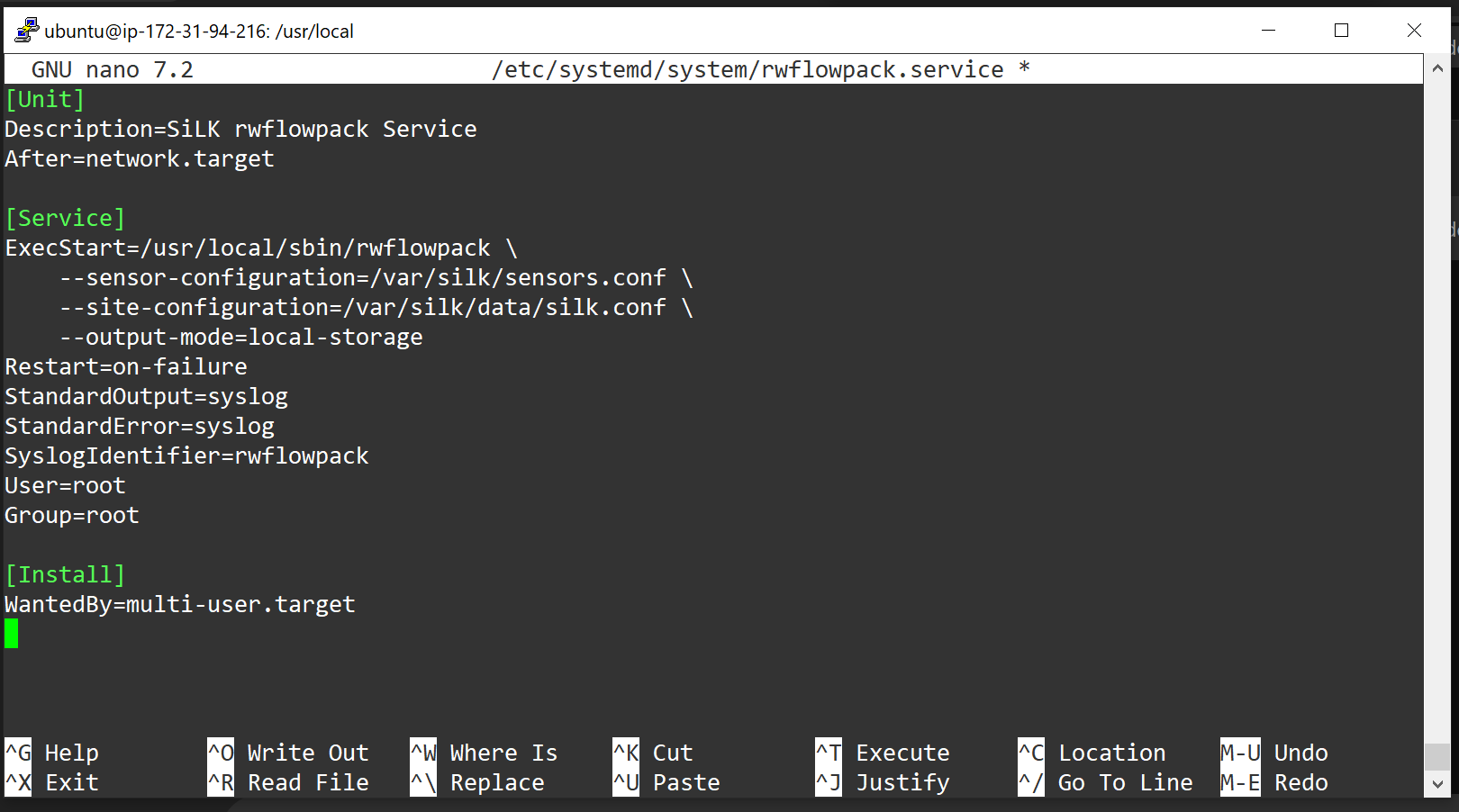
Step – 28 Editing sensors.conf file and after editing it it was looking like this, as ipblocks IP Address was given from my AWS EC2 instance.



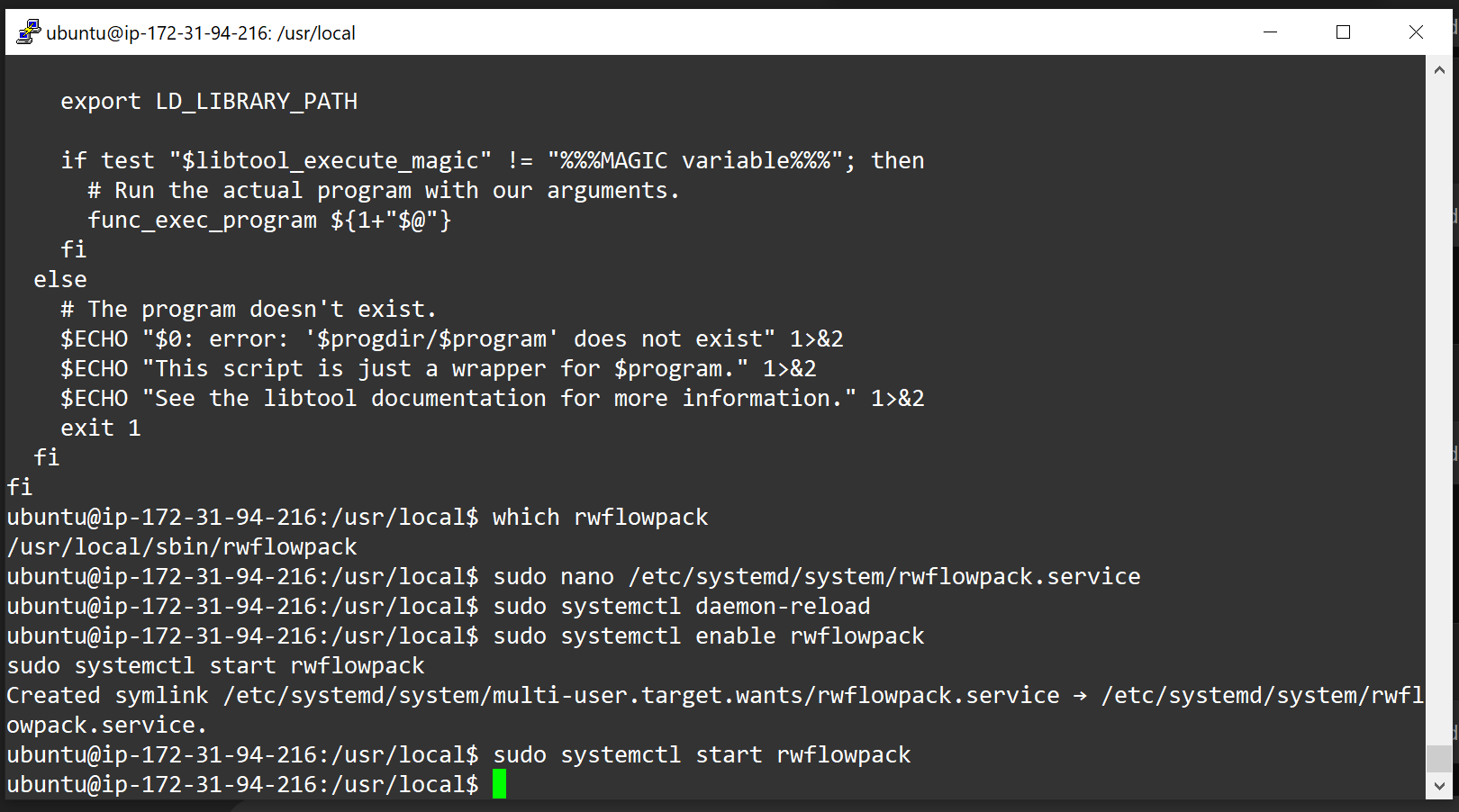
Step – 29 Adding rwflowpack.service in such a manner that it should not fail the service after starting so re-configuring it.



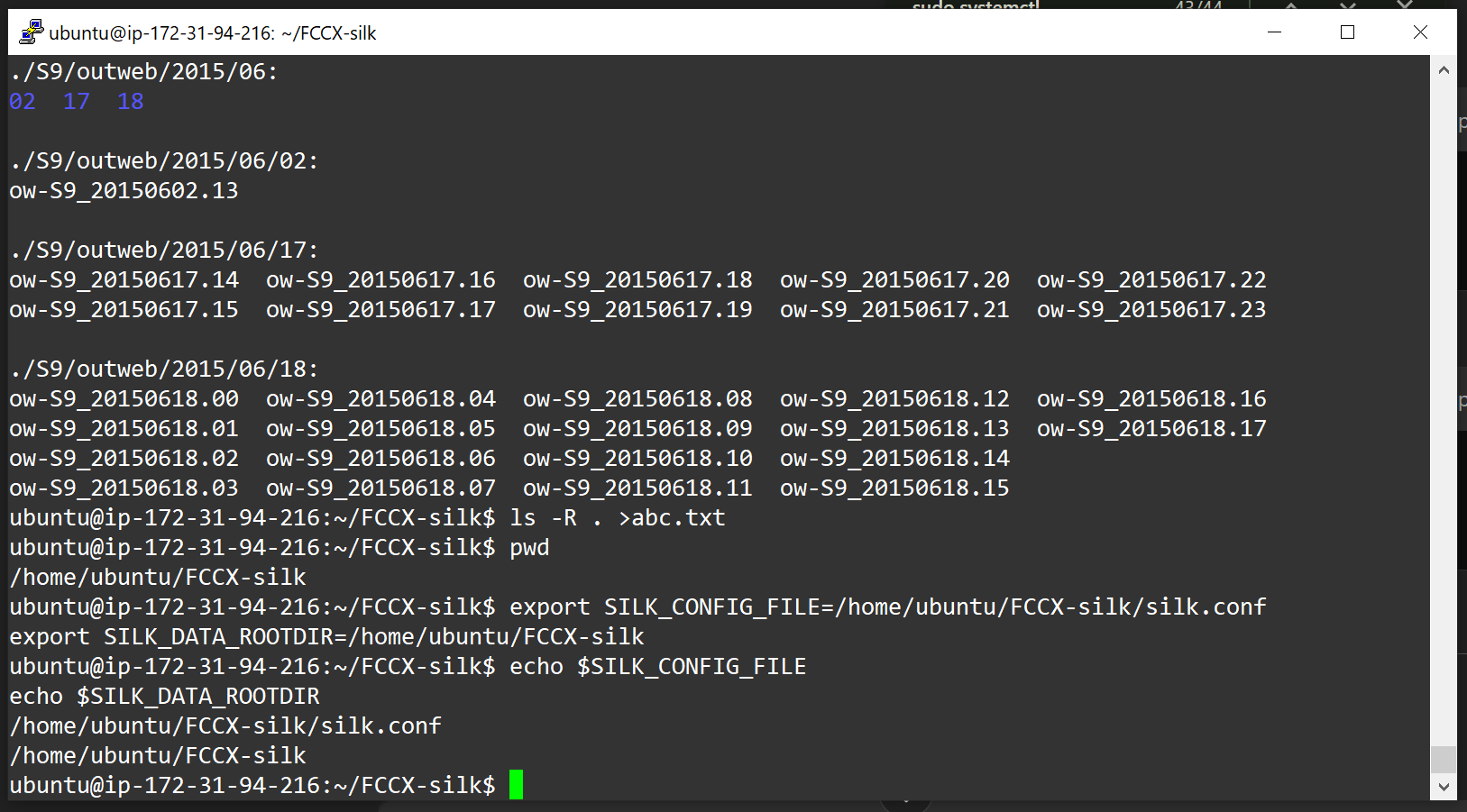
Step – 30 Here is the code for the same.



Step – 31 Here is the code for rwflowpack to start and check its status to ideal or inactive/active.



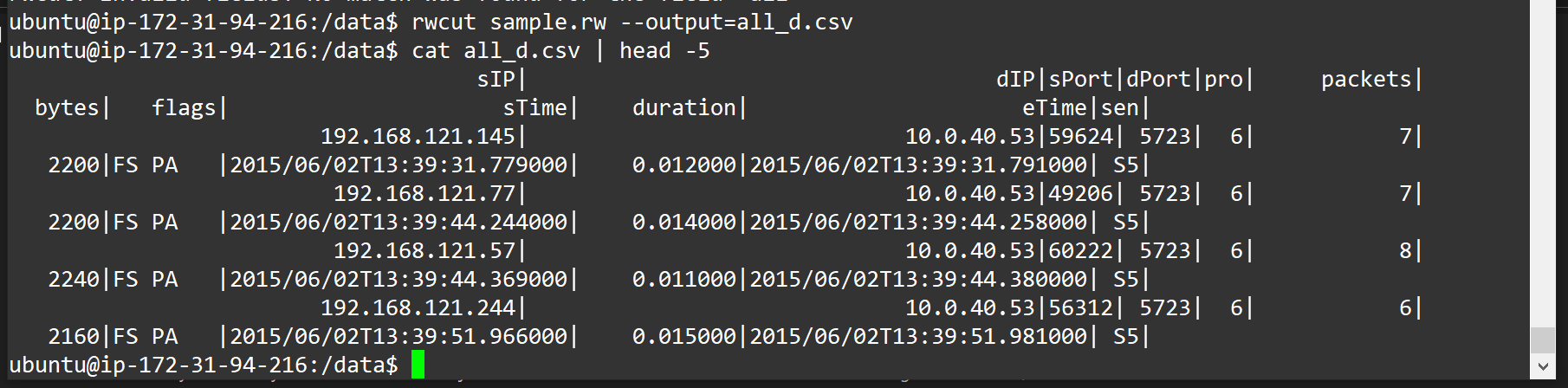
Step –32 Setting up the certain variables such as SILK\_CONFIG\_FILE and SILK\_DATA\_ROOTDIR and do the needful.



Step - 33 Fetching all the records using rwfilter and then store it in filtered.rw. After which, running rwstats command to look at first 5 data from the dataset of FCCX-silk

***(Note: Here I have taken the date from 2000 to 2039 for experimental purpose but actual date lies for 2015 dataset only, but I forgot to take that screenshot, so pasted this screenshot. Also there is no change in the output as said)***

Step - 34 Converting all the data fetched to csv file and then printing the first 5 records using cat command.



1. Classify nodes based on the amount of TCP traffic they process. For instance, you can group nodes that handle TCP traffic within a specific range, such as 10-100 packets per second, into one class. Use any two algorithms, say Very Fast Decision Trees (VFDT) and On Demand Classification, discussed in the lectures to perform this classification. During the demonstration, you should demonstrate the accuracy of your classification by sending a query that retrieves data from at least two different ranges.

Configuring Amazon AWS – S3 for Storing CSV File

Step – 1 Creating Bucket in Amazon S3

A screenshot of a computer

Description automatically generated

Step – 2 Allowing Public Access for this bucket

A screenshot of a computer

Description automatically generated

Step – 3 Creating Bucket with Following Encryption Settings.

A screenshot of a computer

Description automatically generated

Step – 4 Uploading 4.37 GB of CSV File Generated from Silk Suite

A screenshot of a computer

Description automatically generated

Step – 5 Selecting Storage Class as Intelligent Tiering

A screenshot of a computer

Description automatically generated

Step – 6 Clicking on Updload Button for Uploading the CSV File from Local to S3 Bucket.

A screenshot of a computer

Description automatically generated

Step – 7 Creating Amazon S3 Full Access Account for getting AWS Key and , AWS Secret ID and other details.

A screenshot of a computer

Description automatically generated

Step 1: Install Required Libraries

In this step, we install the necessary libraries for data processing, model training, and visualization. These include:

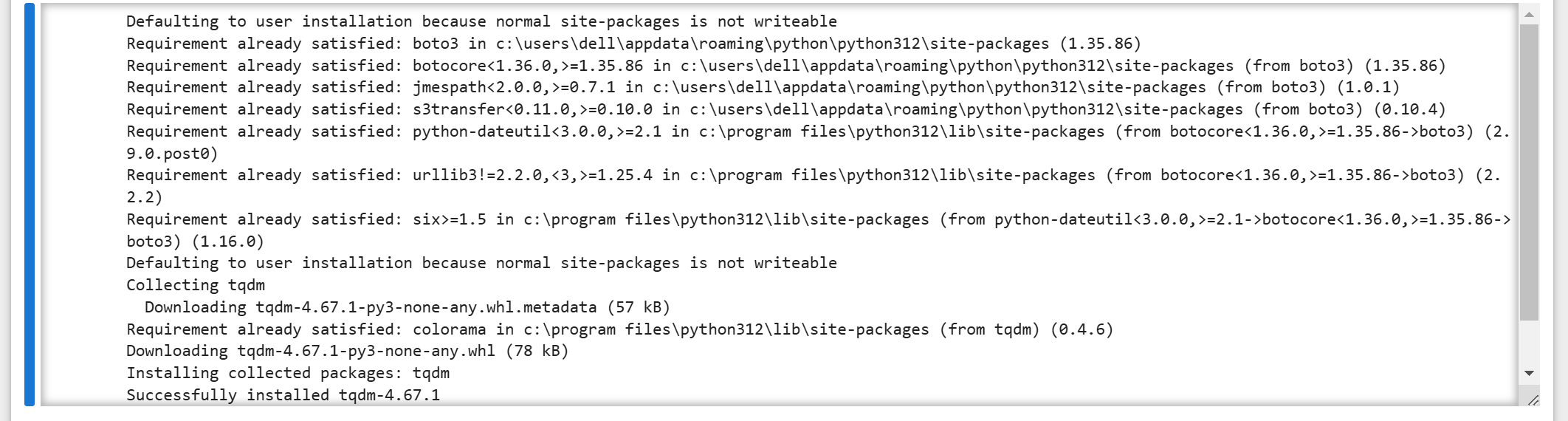
* numpy: A library for numerical computations.
* river: For implementing Very Fast Decision Trees (VFDT) and online machine learning.
* boto3: For accessing AWS S3 to load data.
* tqdm: For displaying progress bars during iterative processes.

!pip install numpy

!pip install river

!pip install boto3

!pip install tqdm



Step 2: Import Libraries and Initialize Global Variables

In this step, we import the necessary libraries for handling data, accessing AWS S3, and managing progress bars during file downloads.

* os: For handling file paths.
* pandas: For loading and processing CSV files into dataframes.
* boto3: For accessing and downloading files from AWS S3.
* StringIO: For handling CSV content in-memory.
* tqdm: For displaying progress bars during file downloads.

We also declare data as a global variable to store the loaded dataset for further processing.

import os

import pandas as pd

import boto3

from io import StringIO

from tqdm import tqdm  # For progress bar

# Declare `data` as a global variable

data = None

Step 3: Define Functions to Load Data

We define two functions to load the dataset:

1. **load\_local\_file()**:
   * Loads a CSV file from the local system using pandas.read\_csv.
   * Displays a preview of the data.
2. **load\_s3\_file\_direct()**:
   * Downloads a CSV file from AWS S3 using the boto3 library.
   * Displays a progress bar while downloading the file.
   * Loads the downloaded CSV into a Pandas dataframe.

**Key Points:**

* The AWS S3 credentials (aws\_access\_key\_id and aws\_secret\_access\_key) and bucket details are hardcoded in this example.
* The function uses a tqdm progress bar to provide feedback during the download.

def load\_local\_file():

    global data  # Declare as global to make it accessible outside the function

    csv\_file\_path = r"C:\Users\dell\Downloads\stream-analytics-all-data.csv"

    if os.path.exists(csv\_file\_path):

        try:

            data = pd.read\_csv(csv\_file\_path, delimiter='|')

            print("Data loaded successfully from local file!")

            print(data.head())

        except Exception as e:

            print(f"Error reading the local file: {e}")

    else:

        print("File does not exist. Please check the path and try again.")

def load\_s3\_file\_direct():

    global data  # Declare as global to make it accessible outside the function

    aws\_access\_key\_id = r"<aws-key-id>"

    aws\_secret\_access\_key = r"<aws-secret-access-key>"

    bucket\_name = "stream-analytics-g23ai2087"

    file\_key = "stream-analytics-all-data.csv"

    try:

        # Create S3 client

        s3 = boto3.client(

            's3',

            aws\_access\_key\_id=aws\_access\_key\_id,

            aws\_secret\_access\_key=aws\_secret\_access\_key,

        )

        # Get the object metadata to determine the file size

        obj\_metadata = s3.head\_object(Bucket=bucket\_name, Key=file\_key)

        file\_size = obj\_metadata['ContentLength']

        # Download the object with a progress bar

        response = s3.get\_object(Bucket=bucket\_name, Key=file\_key)

        # Use a progress bar to monitor the download

        chunk\_size = 10 \* 1024 \* 1024  # 10 MB chunks

        with tqdm(total=file\_size, unit='B', unit\_scale=True, desc="Downloading") as pbar:

            csv\_content = ""

            for chunk in response['Body'].iter\_chunks(chunk\_size):

                csv\_content += chunk.decode('utf-8')

                pbar.update(len(chunk))

        # Load the CSV content directly into pandas DataFrame

        data = pd.read\_csv(StringIO(csv\_content), delimiter='|')

        print("\nData loaded successfully from S3!")

        print(data.head())

    except Exception as e:

        print(f"Error loading file from S3: {e}")

Step 4: Create a Menu for Data Loading

We define a menu() function to allow the user to select how they want to load the dataset:

1. **Load from Local File**: Select option 1 to load a local CSV file.
2. **Load from S3**: Select option 2 to load the CSV file directly from AWS S3.

**Key Points:**

* The menu() function interacts with the user via the command line to get the choice.
* The load\_local\_file() or load\_s3\_file\_direct() function is called based on the user's input.

def menu():

    print("\nMenu:")

    print("1. Load from local CSV file")

    print("2. Load directly from AWS S3")

    choice = input("Enter your choice (1/2): ").strip()

    if choice == '1':

        load\_local\_file()

    elif choice == '2':

        load\_s3\_file\_direct()

    else:

        print("Invalid choice. Please enter 1, 2, or 3.")

Step 5: Run the Menu

Finally, we run the menu() function to allow the user to choose the data loading method. After loading the dataset, the global data variable will contain the loaded DataFrame, which can be used for further analysis and processing.

# Run the menu

menu()

**Output:**

Menu:

1. Load from local CSV file

2. Load directly from AWS S3

Enter your choice (1/2): 1

Data loaded successfully from local file!

sIP \

0 192.168.121.145

1 192.168.121.77

2 192.168.121.57

3 192.168.121.244

4 192.168.121.57

dIP sPort dPort pro packets \

0 10.0.40.53 59624 5723 6 7

1 10.0.40.53 49206 5723 6 7

2 10.0.40.53 60222 5723 6 8

3 10.0.40.53 56312 5723 6 6

4 10.0.40.20 60223 389 6 11

bytes flags sTime duration \

0 2200 FS PA 2015/06/02T13:39:31.779000 0.012

1 2200 FS PA 2015/06/02T13:39:44.244000 0.014

2 2240 FS PA 2015/06/02T13:39:44.369000 0.011

3 2160 FS PA 2015/06/02T13:39:51.966000 0.015

4 2877 FS PA 2015/06/02T13:39:54.712000 0.047

eTime sen Unnamed: 12

0 2015/06/02T13:39:31.791000 S5 NaN

1 2015/06/02T13:39:44.258000 S5 NaN

2 2015/06/02T13:39:44.380000 S5 NaN

3 2015/06/02T13:39:51.981000 S5 NaN

4 2015/06/02T13:39:54.759000 S5 NaN

Step 6: Preview the Loaded Data

In this step, we display the first few rows of the loaded dataset to ensure it has been successfully loaded and is in the correct format.

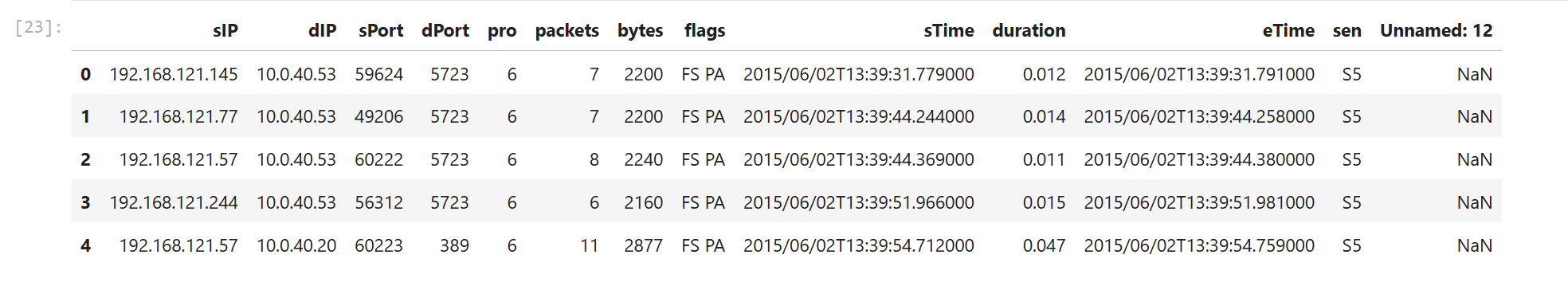
**Key Points:**

* The data variable contains the dataset loaded using the menu options (either from a local file or AWS S3).
* The head() function from Pandas is used to display the first few rows of the DataFrame.

# Display the first few rows to ensure it's loaded correctly

data.head()

**Output:**



Step 7: Explore the Dataset Headers

In this step, we count and list all the headers (columns) in the loaded dataset to understand its structure and ensure the expected fields are present.

**Key Points:**

* **Count Headers**: The len() function is used to count the total number of columns in the dataset.
* **List Headers**: The columns attribute of the Pandas DataFrame provides the list of all column names.

# Count the number of headers

num\_headers = len(data.columns)

print(f"Number of headers (columns): {num\_headers}")

# List all headers

print("Headers in the CSV:")

print(data.columns.tolist())

**Output:**

Number of headers (columns): 13

Headers in the CSV:

[' sIP', ' dIP', 'sPort', 'dPort', 'pro', ' packets', ' bytes', ' flags', ' sTime', ' duration', ' eTime', 'sen', 'Unnamed: 12']

Step 8: Dataset Overview and Summary Statistics

In this step, we examine the overall structure of the dataset and compute summary statistics for the numeric columns.

**Key Points:**

* **data.info()**:
  + Displays a concise summary of the dataset, including:
    - Total number of rows and columns.
    - Data types of each column.
    - Number of non-null values for each column.
* **data.describe()**:
  + Provides summary statistics (e.g., count, mean, standard deviation, min, max) for all numeric columns in the dataset.
  + Helps in understanding the distribution and range of numeric variables.

# Display data information

data.info()

# Get a summary of numeric columns

data.describe()

**Output:**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 21216409 entries, 0 to 21216408

Data columns (total 13 columns):

# Column Dtype

--- ------ -----

0 sIP object

1 dIP object

2 sPort int64

3 dPort int64

4 pro int64

5 packets int64

6 bytes int64

7 flags object

8 sTime object

9 duration float64

10 eTime object

11 sen object

12 Unnamed: 12 float64

dtypes: float64(2), int64(5), object(6)

memory usage: 2.1+ GB

A screenshot of a computer screen

Description automatically generated

Step 9: Filter Data for TCP Protocol

In this step, we filter the dataset to include only rows where the protocol is TCP. This is done by checking if the pro column has a value of 6, which represents the TCP protocol.

**Key Points:**

* **Filtering**:
  + The data DataFrame is filtered using a condition data['pro'] == 6.
  + The filtered dataset is stored in a new DataFrame named tcp\_data.
* **Copy of Filtered Data**:
  + We use the .copy() method to avoid potential warnings about modifying the original DataFrame.
* **Preview**:
  + The first few rows of the filtered dataset are displayed using the head() function.

# Filter rows where protocol is TCP (pro == 6)

tcp\_data = data[data['pro'] == 6].copy()

# Display the filtered data

print(tcp\_data.head())

**Output:**

sIP \

0 192.168.121.145

1 192.168.121.77

2 192.168.121.57

3 192.168.121.244

4 192.168.121.57

dIP sPort dPort pro packets \

0 10.0.40.53 59624 5723 6 7

1 10.0.40.53 49206 5723 6 7

2 10.0.40.53 60222 5723 6 8

3 10.0.40.53 56312 5723 6 6

4 10.0.40.20 60223 389 6 11

bytes flags sTime duration \

0 2200 FS PA 2015/06/02T13:39:31.779000 0.012

1 2200 FS PA 2015/06/02T13:39:44.244000 0.014

2 2240 FS PA 2015/06/02T13:39:44.369000 0.011

3 2160 FS PA 2015/06/02T13:39:51.966000 0.015

4 2877 FS PA 2015/06/02T13:39:54.712000 0.047

eTime sen Unnamed: 12

0 2015/06/02T13:39:31.791000 S5 NaN

1 2015/06/02T13:39:44.258000 S5 NaN

2 2015/06/02T13:39:44.380000 S5 NaN

3 2015/06/02T13:39:51.981000 S5 NaN

4 2015/06/02T13:39:54.759000 S5 NaN

Step 10: Clean Column Names in Filtered Data

In this step, we clean the column names of the filtered TCP dataset (tcp\_data) to remove any leading or trailing spaces. Additionally, we rename specific columns for consistent naming.

**Key Points:**

* **Strip Leading/Trailing Spaces**:
  + The str.strip() function is applied to tcp\_data.columns to remove unnecessary spaces in column names.
* **Rename Columns**:
  + Specific columns are renamed for better readability and consistency using the rename() function with inplace=True.
* **Verification**:
  + The column names are displayed before and after cleaning to confirm the changes.

print(tcp\_data.columns.tolist())

# Strip leading/trailing spaces from column names

tcp\_data.columns = tcp\_data.columns.str.strip()

# Verify again

print(tcp\_data.columns.tolist())

tcp\_data.rename(columns={' duration ': 'duration'}, inplace=True)

**Output:**

[' sIP', ' dIP', 'sPort', 'dPort', 'pro', ' packets', ' bytes', ' flags', ' sTime', ' duration', ' eTime', 'sen', 'Unnamed: 12']

['sIP', 'dIP', 'sPort', 'dPort', 'pro', 'packets', 'bytes', 'flags', 'sTime', 'duration', 'eTime', 'sen', 'Unnamed: 12']

Step 11: Calculate Packets Per Second

In this step, we calculate a new column, packets\_per\_second, for the TCP dataset. This column represents the rate of packets handled per second for each row.

**Key Points:**

* **Calculation**:
  + The formula for packets\_per\_second is: packets\_per\_second = packets/duration
  + If duration is 0, the value is set to 0 to avoid division by zero.
* **Lambda Function**:
  + A lambda function is applied to each row of the DataFrame using the apply() method with axis=1.
* **Preview**:
  + The first few rows of the dataset, including the new column, are displayed to verify the calculation.

tcp\_data['packets\_per\_second'] = tcp\_data.apply(

    lambda row: row['packets'] / row['duration'] if row['duration'] > 0 else 0,

    axis=1

)

# Preview the results

print(tcp\_data[['sIP', 'packets', 'duration', 'packets\_per\_second']].head())

**Output:**

sIP packets duration \

0 192.168.121.145 7 0.012

1 192.168.121.77 7 0.014

2 192.168.121.57 8 0.011

3 192.168.121.244 6 0.015

4 192.168.121.57 11 0.047

packets\_per\_second

0 583.333333

1 500.000000

2 727.272727

3 400.000000

4 234.042553

Step 12: Train and Test Models for TCP Traffic Classification

In this step, we perform the following tasks to classify TCP traffic rates based on the calculated packets\_per\_second:

1. **Ensure Valid Duration Values**:
   * Replace 0 values in the duration column with 1 to avoid division by zero.
2. **Calculate packets\_per\_second**:
   * Compute the traffic rate as packets divided by duration for each row in the dataset.
3. **Define Traffic Classes**:
   * Create a new column traffic\_class to group traffic rates into the following categories:
     + Very Low: 0-10 packets/sec
     + Low: 10-100 packets/sec
     + Medium: 100-1000 packets/sec
     + High: Above 1000 packets/sec
4. **Define Features and Target**:
   * Define X (features) as the packets\_per\_second column and y (target) as the traffic\_class column.
5. **Split the Dataset**:
   * Split the dataset into training and testing sets using an 80-20 split, stratifying by traffic\_class.
6. **Handle Missing Values**:
   * Use a SimpleImputer to replace missing values with the mean of the respective columns.
7. **Train Very Fast Decision Tree (VFDT)**:
   * Incrementally train the VFDT model using the river library on the training data.
8. **Predict with VFDT**:
   * Use the trained VFDT model to predict traffic classes on the test set and display a classification report.
9. **Train K-Nearest Neighbors (KNN)**:
   * Train the KNN classifier with 3 neighbors using the training data.
10. **Predict with KNN**:
    * Use the trained KNN model to predict traffic classes on the test set and display a classification report.
11. **Query Results for Specific Ranges**:
    * Query both VFDT and KNN models for traffic rates of 50 packets/sec and 500 packets/sec to demonstrate their predictions.

**Key Points:**

* **VFDT**:
  + Handles incremental learning, suitable for online data streams.
* **KNN**:
  + Performs lazy evaluation, classifying data based on nearest neighbors.
* **Evaluation**:
  + Classification reports provide precision, recall, F1-score, and support for each class.

import pandas as pd

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

from sklearn.impute import SimpleImputer

from river.tree import HoeffdingTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report, accuracy\_score

# Step 1: Load the dataset (assuming tcp\_data is already loaded)

# Ensure duration values are valid to avoid division by zero

tcp\_data['duration'] = tcp\_data['duration'].replace(0, 1)

# Step 2: Calculate packets\_per\_second if not already present

tcp\_data['packets\_per\_second'] = tcp\_data['packets'] / tcp\_data['duration']

# Step 3: Define the traffic class based on packets\_per\_second

tcp\_data['traffic\_class'] = pd.cut(

    tcp\_data['packets\_per\_second'],

    bins=[0, 10, 100, 1000, float('inf')],

    labels=['Very Low', 'Low', 'Medium', 'High']

)

# Step 4: Define features (X) and target (y)

X = tcp\_data[['packets\_per\_second']].copy()

y = tcp\_data['traffic\_class']

# Step 5: Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

# Handle missing values (if any)

imputer = SimpleImputer(strategy='mean')

X\_train = imputer.fit\_transform(X\_train)

X\_test = imputer.transform(X\_test)

# Step 6: Train the VFDT (HoeffdingTreeClassifier)

vfdt = HoeffdingTreeClassifier()

for i in range(len(X\_train)):

    x = {'packets\_per\_second': X\_train[i][0]}

    vfdt.learn\_one(x, str(y\_train.iloc[i]))

# Predict using VFDT

vfdt\_predictions = [vfdt.predict\_one({'packets\_per\_second': X\_test[i][0]}) for i in range(len(X\_test))]

y\_test\_str = y\_test.astype(str)  # Convert y\_test to string for comparison

print("VFDT Classification Report:")

print(classification\_report(y\_test\_str, vfdt\_predictions))

# Step 7: Train KNN Classifier

knn = KNeighborsClassifier(n\_neighbors=3)

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

# Predict using KNN

knn\_predictions = knn.predict(X\_test)

print("KNN Classification Report:")

print(classification\_report(y\_test, knn\_predictions))

# Step 8: Query results for specific ranges

query1 = pd.DataFrame([[50]], columns=['packets\_per\_second'])

query2 = pd.DataFrame([[500]], columns=['packets\_per\_second'])

print(f"Query 1 (50 packets/sec) - VFDT: {vfdt.predict\_one({'packets\_per\_second': 50})}")

print(f"Query 2 (500 packets/sec) - VFDT: {vfdt.predict\_one({'packets\_per\_second': 500})}")

print(f"Query 1 (50 packets/sec) - KNN: {knn.predict(query1)}")

print(f"Query 2 (500 packets/sec) - KNN: {knn.predict(query2)}")

Output:

VFDT Classification Report:

precision recall f1-score support

High 1.00 1.00 1.00 95743

Low 1.00 1.00 1.00 152509

Medium 1.00 1.00 1.00 691047

Very Low 1.00 1.00 1.00 439027

accuracy 1.00 1378326

macro avg 1.00 1.00 1.00 1378326

weighted avg 1.00 1.00 1.00 1378326

KNN Classification Report:

precision recall f1-score support

High 1.00 1.00 1.00 95743

Low 1.00 1.00 1.00 152509

Medium 1.00 1.00 1.00 691047

Very Low 1.00 1.00 1.00 439027

accuracy 1.00 1378326

macro avg 1.00 1.00 1.00 1378326

weighted avg 1.00 1.00 1.00 1378326

Query 1 (50 packets/sec) - VFDT: Low

Query 2 (500 packets/sec) - VFDT: Medium

Query 1 (50 packets/sec) - KNN: ['Low']

Query 2 (500 packets/sec) - KNN: ['Medium']

Step 13: Visualize Traffic Class Distribution

In this step, we create a bar plot to visualize the distribution of the traffic classes (traffic\_class) in the TCP dataset.

**Key Points:**

* **Value Counts**:
  + The value\_counts() function is used to count the occurrences of each traffic class.
* **Bar Plot**:
  + A bar plot is created using the plot(kind='bar') function in Pandas.
* **Visualization**:
  + The plot includes a title, labeled axes, and displays the number of nodes in each traffic class.

**Purpose:**

* This visualization helps understand the distribution of traffic classes, enabling us to identify any imbalances in the dataset.

import matplotlib.pyplot as plt

# Plot traffic class distribution

tcp\_data['traffic\_class'].value\_counts().plot(kind='bar')

plt.title("Traffic Class Distribution")

plt.xlabel("Traffic Class")

plt.ylabel("Number of Nodes")

plt.show()

**Output:**

A graph of a number of blue bars

Description automatically generated

Step 14: Display Training and Testing Dataset Sizes

In this step, we display the number of samples in the training and testing datasets after splitting the original dataset.

**Key Points:**

* **Training Samples**:
  + The size of the training dataset is determined using the len() function on X\_train.
* **Testing Samples**:
  + The size of the testing dataset is determined using the len() function on X\_test.
* **Purpose**:
  + This step ensures that the dataset is split correctly, and both training and testing datasets have the expected number of samples.

**Output:**

* The number of training and testing samples is printed for verification.

print(f"Number of training samples: {len(X\_train)}")

print(f"Number of test samples: {len(X\_test)}")

**Output:**

Number of training samples: 5513304

Number of test samples: 1378326

Step 15: Train and Test Optimized KNN Classifier

In this step, we train and test an optimized K-Nearest Neighbors (KNN) classifier on the TCP dataset to predict traffic classes.

**Key Points:**

1. **Initialization**:
   * The KNN model is initialized with the following parameters:
     + n\_neighbors=3: Considers the 3 nearest neighbors for classification.
     + algorithm='ball\_tree': Uses a Ball Tree data structure for efficient neighbor searches.
     + n\_jobs=-1: Utilizes all available CPU cores for parallel computation.
2. **Training**:
   * The KNN model is trained using the fit() method on the training dataset (X\_train and y\_train).
3. **Testing on a Subset**:
   * A subset of the test dataset (first 100 samples) is used to evaluate the model for faster computation.
4. **Predictions**:
   * Predictions are made using the predict() method on the subset of test data (subset\_X\_test).
5. **Accuracy Score**:
   * The accuracy\_score() function computes the classification accuracy, comparing the predicted values with the true labels (subset\_y\_test).

**Purpose:**

* To evaluate the performance of the optimized KNN classifier in predicting traffic classes.

**Output:**

* The accuracy of the KNN model on the subset of test data is printed.

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Initialize KNN with optimizations

knn = KNeighborsClassifier(n\_neighbors=3, algorithm='ball\_tree', n\_jobs=-1)

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

# Use a subset of X\_test

subset\_X\_test = X\_test[:100]

subset\_y\_test = y\_test[:100]

# Predict on the subset

knn\_predictions = knn.predict(subset\_X\_test)

knn\_accuracy = accuracy\_score(subset\_y\_test, knn\_predictions)

print(f"KNN Accuracy (optimized): {knn\_accuracy}")

**Output:**

KNN Accuracy (optimized): 1.0

1. Detect the anomaly in the fetched data. Anomalies are defined as the nodes which send large number of TCP packets per sec, say 1K per sec. You are free to define your own threshold based on your observation in the given data stream.

Step 1: Identify and Display Anomalous Nodes

In this step, we identify nodes with anomalous TCP traffic based on a predefined threshold for packets\_per\_second. Anomalous nodes are then displayed for further analysis.

**Key Points:**

1. **Anomaly Threshold**:
   * Define a threshold value (anomaly\_threshold = 1000) for identifying anomalies in traffic rates.
2. **Filter Anomalous Nodes**:
   * Filter rows in the dataset where the packets\_per\_second value exceeds the defined threshold.
3. **Display Anomalous Nodes**:
   * Display the source IP (sIP), destination IP (dIP), and packets\_per\_second values for the detected anomalies.

**Purpose:**

* To detect and analyze nodes with unusually high TCP traffic rates, which may indicate abnormal behavior or events.

**Output:**

* A list of anomalous nodes exceeding the threshold, along with their key attributes.

# Define the threshold for anomalies

anomaly\_threshold = 1000

# Identify anomalous nodes

anomalies = tcp\_data[tcp\_data['packets\_per\_second'] > anomaly\_threshold]

# Display anomalous nodes

print(f"Anomalies detected (packets\_per\_second > {anomaly\_threshold}):")

print(anomalies[['sIP', 'dIP', 'packets\_per\_second']])

**Output:**

Anomalies detected (packets\_per\_second > 1000):

sIP \

1004 192.168.141.189

1451 192.168.40.92

1501 192.168.40.25

1773 192.168.40.20

3650 192.168.40.51

... ...

21216399 10.0.40.21

21216400 10.0.40.21

21216401 10.0.40.21

21216403 10.0.40.21

21216406 10.0.40.21

dIP packets\_per\_second

1004 67.215.0.8 2000.000000

1451 10.0.40.23 1500.000000

1501 10.0.40.20 2000.000000

1773 192.168.143.239 1666.666667

3650 10.0.40.20 2000.000000

... ... ...

21216399 10.0.20.59 2500.000000

21216400 10.0.20.59 2500.000000

21216401 10.0.20.59 2500.000000

21216403 10.0.20.59 2500.000000

21216406 10.0.20.58 3000.000000

[478716 rows x 3 columns]

Step 2: Count and Analyze Anomalies

In this step, we count the total number of anomalous nodes and analyze the distribution of their packets\_per\_second values.

**Key Points:**

1. **Count Anomalies**:
   * Use the shape[0] attribute to get the total number of rows (anomalies) in the filtered dataset (anomalies).
2. **Analyze Distribution**:
   * Use the describe() method to generate summary statistics for the packets\_per\_second column of the anomalous nodes.
   * Key statistics include:
     + Count: Total number of anomalies.
     + Mean: Average packets\_per\_second value for anomalies.
     + Standard Deviation (std): Spread of the values.
     + Min/Max: Minimum and maximum traffic rates for anomalies.
     + Percentiles (25%, 50%, 75%): Quartiles of the data distribution.

**Purpose:**

* To quantify and analyze the traffic characteristics of the identified anomalies.

**Output:**

* Total number of anomalies detected.
* Summary statistics for the packets\_per\_second column in the anomalous data.

# Count the number of anomalies

num\_anomalies = anomalies.shape[0]

print(f"Number of anomalies detected: {num\_anomalies}")

# Analyze distribution of packets\_per\_second

print("Summary statistics for anomalies:")

print(anomalies['packets\_per\_second'].describe())

**Output:**

Number of anomalies detected: 478716

Summary statistics for anomalies:

count 478716.000000

mean 1512.644088

std 508.879721

min 1000.000000

25% 1250.000000

50% 1333.333333

75% 1500.000000

max 18899.478293

Name: packets\_per\_second, dtype: float64

Step 3: Visualize Packets Per Second (Normal vs Anomalous Nodes)

In this step, we create a scatter plot to visualize the packets\_per\_second values for all nodes, distinguishing between normal and anomalous nodes.

**Key Points:**

1. **Scatter Plot**:
   * Each point represents a node with its source IP (sIP) on the x-axis and its packets\_per\_second value on the y-axis.
2. **Normal Nodes**:
   * Plotted in the default color (blue) to represent nodes with traffic below the anomaly threshold.
3. **Anomalous Nodes**:
   * Highlighted in red to indicate nodes with traffic exceeding the anomaly threshold.
4. **Threshold Line**:
   * A horizontal line (axhline) is added to represent the anomaly threshold (anomaly\_threshold).
5. **Customization**:
   * Labels, title, and legend are added for clarity.
   * Source IPs are rotated for better readability on the x-axis.

**Purpose:**

* To visually distinguish between normal and anomalous nodes based on their packets\_per\_second values.

**Output:**

* A scatter plot showing the distribution of normal and anomalous nodes.

import matplotlib.pyplot as plt

# Scatter plot of packets\_per\_second

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

plt.scatter(tcp\_data['sIP'], tcp\_data['packets\_per\_second'], label="Normal")

plt.scatter(anomalies['sIP'], anomalies['packets\_per\_second'], color='red', label="Anomalies")

plt.axhline(y=anomaly\_threshold, color='orange', linestyle='--', label="Threshold")

plt.title("Packets per Second: Normal vs Anomalous Nodes")

plt.xlabel("Source IP")

plt.ylabel("Packets per Second")

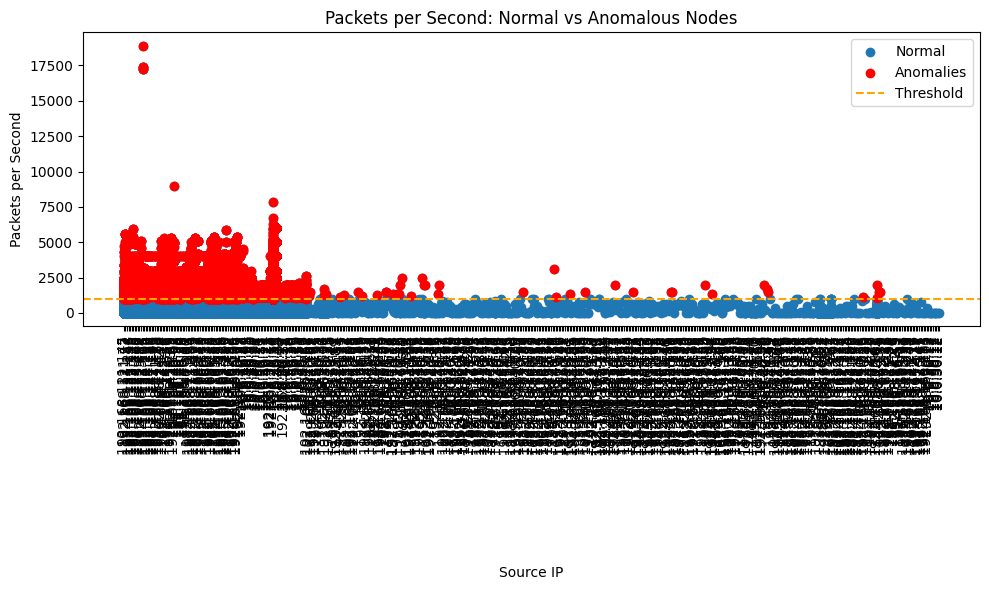
plt.legend()

plt.xticks(rotation=90)

plt.tight\_layout()

plt.show()

**Output:**



Step 4 : Visualize Packets Per Second by IP Group (Normal vs Anomalous)

In this step, we group the TCP data by the first two segments of the IP address (e.g., 192.168) and analyze the average packets\_per\_second within each group. The results are visualized in a bar chart, distinguishing between normal and anomalous groups.

**Key Points:**

1. **Group by IP**:
   * The ip\_group column is created by extracting the first two segments of the source IP (sIP).
   * Data is grouped by ip\_group, and the average packets\_per\_second is calculated for each group.
2. **Identify Anomalous Groups**:
   * Groups with an average packets\_per\_second exceeding the anomaly threshold (anomaly\_threshold = 1000) are marked as anomalous using a new column is\_anomaly.
3. **Bar Plot**:
   * Normal groups are plotted in blue.
   * Anomalous groups are highlighted in red.
   * A horizontal orange line represents the anomaly threshold.
4. **Customization**:
   * The bar chart includes labels, a title, and a legend for clear interpretation.
   * The x-axis labels (IP groups) are rotated for readability.

**Purpose:**

* To identify and visualize patterns in traffic behavior across different IP groups, highlighting groups with anomalous traffic rates.

**Output:**

* A bar chart showing average packets\_per\_second for each IP group, distinguishing between normal and anomalous groups.

import pandas as pd

import matplotlib.pyplot as plt

# Group the data by the first two segments of the IP address (e.g., '192.168')

tcp\_data['ip\_group'] = tcp\_data['sIP'].apply(lambda x: '.'.join(x.split('.')[:2]))

# Aggregate packets per second by group

grouped\_data = tcp\_data.groupby('ip\_group')['packets\_per\_second'].mean().reset\_index()

# Identify anomalous groups

anomaly\_threshold = 1000  # Example threshold for anomalies

grouped\_data['is\_anomaly'] = grouped\_data['packets\_per\_second'] > anomaly\_threshold

# Plot the grouped data

plt.figure(figsize=(12, 8))

# Normal groups

normal\_groups = grouped\_data[~grouped\_data['is\_anomaly']]

plt.bar(

    normal\_groups['ip\_group'],

    normal\_groups['packets\_per\_second'],

    color='blue',  # Normal group color

    label="Normal",

)

# Anomalous groups

anomalous\_groups = grouped\_data[grouped\_data['is\_anomaly']]

plt.bar(

    anomalous\_groups['ip\_group'],

    anomalous\_groups['packets\_per\_second'],

    color='red',  # Anomalous group color

    label="Anomalies",

)

# Add a threshold line

plt.axhline(

    y=anomaly\_threshold,

    color='orange',

    linestyle='--',

    label=f"Threshold ({anomaly\_threshold} packets/sec)",

)

# Title and labels

plt.title("Packets per Second by IP Group: Normal vs Anomalous", fontsize=16)

plt.xlabel("IP Group (e.g., 192.168.\*)", fontsize=14)

plt.ylabel("Average Packets per Second", fontsize=14)

# Rotate x-axis labels for readability

plt.xticks(rotation=45, fontsize=10)

# Add legend

plt.legend(loc="upper right", fontsize=12)

# Adjust layout

plt.tight\_layout()

# Show the plot

plt.show()

**Output:**

A graph of blue bars

Description automatically generated

Step 5: Group and Count Anomalies by Source IP

In this step, we group the anomalous nodes by their source IP (sIP) and count the number of anomalies for each source IP.

**Key Points:**

1. **Group by Source IP**:
   * Use the groupby() function to group the anomalies by the sIP column.
2. **Count Anomalies**:
   * Use the size() function to count the number of anomalies for each source IP.
   * Reset the index of the resulting DataFrame to create a new column anomaly\_count that stores the counts.
3. **Sort Results**:
   * The results are sorted in descending order of the anomaly\_count column to identify the most frequently anomalous source IPs.
4. **Output**:
   * A list of source IPs with their corresponding anomaly counts is displayed.

**Purpose:**

* To identify and quantify the nodes (source IPs) contributing to anomalous traffic patterns.

**Output:**

* A sorted list of source IPs with their anomaly counts, starting from the most anomalous.

# Group by source IP and count anomalies

anomalies\_by\_sIP = anomalies.groupby('sIP').size().reset\_index(name='anomaly\_count')

print("Anomalies by Source IP:")

print(anomalies\_by\_sIP.sort\_values(by='anomaly\_count', ascending=False))

**Output:**

Anomalies by Source IP:

sIP anomaly\_count

3 10.0.40.21 52182

10 10.0.40.53 50821

44 192.168.70.10 26882

1 10.0.20.59 25355

34 192.168.40.20 17370

.. ... ...

143 10.0.20.59 1

144 192.168.121.77 1

145 192.168.163.154 1

146 192.168.50.12 1

147 192.168.70.10 1

[148 rows x 2 columns]

Step 6: Calculate Total and Percentage of Anomalous Packets

In this step, we calculate the total number of packets handled by anomalous nodes and determine their percentage contribution to the overall packets in the dataset.

**Key Points:**

1. **Total Anomalous Packets**:
   * Use the sum() function on the packets column of the anomalies DataFrame to calculate the total number of packets handled by anomalous nodes.
2. **Percentage of Anomalous Packets**:
   * Calculate the percentage of packets handled by anomalous nodes relative to the total packets in the dataset: percentage = (total anomalous packets) / total packets in dataset) \* 100
3. **Output**:
   * The percentage of anomalous packets is displayed, rounded to two decimal places.

**Purpose:**

* To quantify the impact of anomalous nodes on the overall traffic in the dataset.

**Output:**

* The total number of packets handled by anomalous nodes.
* The percentage of anomalous packets relative to the total traffic.

# Calculate total packets for anomalies

total\_anomalous\_packets = anomalies['packets'].sum()

# Calculate percentage of anomalous packets

percentage\_anomalous\_packets = (total\_anomalous\_packets / tcp\_data['packets'].sum()) \* 100

print(f"Percentage of anomalous packets: {percentage\_anomalous\_packets:.2f}%")

**Output:**

Percentage of anomalous packets: 9.67%