**📘 Project Documentation**

**1. Project Overview**

This project aims to build a real-time monitoring, anomaly detection and traffic analysis system using the **Isolation Forest** algorithm. The system processes network traffic data, detects anomalies, and visualizes these anomalies with insightful charts and reports. Key features include anomaly detection, real-time alerts, traffic flow analysis, and network interaction visualization.

**Key Features**

* Real-time anomaly detection using the Isolation Forest algorithm.
* Visualizations including time series plots, boxplots, heatmaps, radar charts, Sankey diagrams, and interactive network graphs.
* Anomaly alerts based on configurable thresholds.
* IP traffic flow analysis and anomaly profiling.

**Problem Statement**

The objective of this project is to detect anomalies in network traffic in real-time, understand the traffic patterns, and identify potential malicious behavior. By using statistical and machine learning methods like **Isolation Forest**, the system identifies outliers (anomalies) in the traffic dataset based on selected features.

**Use Case**

* **Network Security**: Detecting unusual traffic behavior that could indicate network intrusions or DDoS attacks.
* **Data Analysis**: Understanding the flow of data between source and destination IPs.
* **Anomaly Detection**: Profiling network behavior and identifying potential issues early.

**Tools and Technologies**

* **Programming Language**: Python
* **Libraries**:
  + pandas: Data manipulation and cleaning.
  + scikit-learn: Machine learning, specifically Isolation Forest.
  + plotly: Data visualization (time series, scatter plots, box plots, heatmaps).
  + networkx, pyvis: Network graph creation.
  + numpy: Numerical operations.
  + scipy: Statistical operations for KDE (Kernel Density Estimation).
  + smtplib: For email alerts.
  + streamlit: Web application framework for interactive dashboard.

**Data Preprocessing**

The dataset includes network traffic data with features such as:

* startDate: Timestamp of the traffic.
* sPackets: Sent packets.
* rPackets: Received packets.
* sBytesSum: Sum of sent bytes.
* rBytesSum: Sum of received bytes.
* sLoad: Sent load.
* rLoad: Received load.
* sIPs: Source IPs.
* rIPs: Destination IPs.

**Data Cleaning**

* Handling missing values by filling them with zeros (df.fillna(0)).
* Converting timestamps to datetime format for time-based analysis.

**🧱 2. Architecture and Modules**

**📦 List of Main Modules:**

| **Module Number** | **Module Name** | **Description** |
| --- | --- | --- |
| 1 | Data Loading & Preprocessing | Ingest and clean the dataset |
| 2 | Anomaly Detection | Use Isolation Forest to detect anomalies |
| 3 | Visualization Dashboard | Streamlit-based interface with multiple tabs |
| 4 | Interactive Charts | Various data plots for deeper understanding |
| 5 | IP Lookup Tool | Lookup behavior and stats for individual IPs |
| 6 | Graphical Flow Visualization | Sankey and Network Graphs to show IP relationships |
| 7 | Rolling Anomaly Rate | Time-series visualization for anomaly rates |

**🧩 3. Module-wise Explanation**

**✅ Module 1: Data Loading & Preprocessing**

**🎯 Objective**

This module is responsible for importing the raw network traffic data, cleaning it, and preparing it for anomaly detection and visualization. It ensures the data is in a structured format with meaningful derived features.

**📌 Key Steps**

1. **Data Import**

The raw dataset, typically in CSV format, is loaded into memory. One of the primary columns in the dataset is startDate, which represents the timestamp of the traffic record.

1. **Datetime Conversion**

The startDate is converted into Python’s datetime format to enable easy extraction of temporal features like hour, day of the week, and date-based filtering.

1. **Data Cleaning**

The dataset is scanned for any missing or null values. If present, these rows are either dropped or imputed based on the design needs. Additionally, redundant or irrelevant columns, if any, are removed to streamline the dataset.

1. **Feature Engineering**

New features are generated from existing data to enrich the dataset. These include:

* + **Hour of the day**: Helps identify peak traffic periods.
  + **Day of the week**: Useful for spotting anomalies based on weekly patterns.
  + **Total Bytes**: A combined feature from incoming and outgoing byte sums to reflect overall data usage.

1. **Optional Normalization or Scaling**

Depending on the anomaly detection model used later, numeric features may be normalized to bring them to a common scale. This step enhances the performance and accuracy of the algorithms.

**🧾 Output**

At the end of this module, the dataset is:

* Clean, consistent, and enriched with new features.
* Ready for further processing in anomaly detection and visualization modules.

**✅ Module 2: Anomaly Detection**

**🎯 Objective**

This module is designed to detect unusual patterns or outliers in the network traffic data. These anomalies can indicate potential threats, system misconfigurations, or usage spikes that need attention.

**📌 Key Concepts**

1. **Understanding Normal vs Abnormal Behavior**

The goal is to identify traffic patterns that deviate significantly from the norm. These deviations could be:

* + Sudden spikes in data usage
  + Unusual access times (e.g., late night traffic)
  + Large file transfers during off-peak hours
  + Consistently high usage by specific users or IPs

1. **Defining an Anomaly**

An anomaly is defined based on a threshold or statistical deviation. For instance, any data point significantly higher than the mean might be considered anomalous.

**⚙️ Approach Used: Statistical Thresholding**

* **Daily Aggregation**:

Network traffic is grouped by each day, summing up total bytes transferred.

* **Mean and Standard Deviation Calculation**:

The average (mean) and standard deviation of daily totals are calculated.

* **Anomaly Criteria**:

A threshold is set (e.g., 2.5× standard deviation above the mean). Any day where the traffic exceeds this threshold is flagged as anomalous.

**🧠 Benefits of This Method**

* **Simplicity & Interpretability**: Easy to understand and explain to non-technical stakeholders.
* **Baseline Insight**: Serves as a good first-pass method for anomaly detection.
* **Low Computation Cost**: Efficient and lightweight, suitable for smaller datasets or real-time alerts.

**🧾 Output**

* A list of days (or data points) that are considered anomalous based on the defined threshold.
* These flagged dates are passed to the visualization module for representation.

**📦 Module 3: Visualization**

**🔍 Overview**

The **Visualization module** is a crucial part of the AI-based network intrusion detection system. It plays a significant role in converting raw numerical data and detection results into interpretable visual formats. This module empowers security analysts, developers, and even non-technical stakeholders to quickly comprehend trends, identify suspicious activities, and make informed decisions based on data insights.

While the anomaly detection module provides the "what", the visualization module provides the "how and why" by visually articulating when and how anomalies occurred in the network traffic. It bridges the gap between AI outputs and human interpretation.

**🎯 Goals and Objectives**

* **Present network traffic trends** in a human-friendly manner.
* **Highlight anomalies** using distinct visual cues to easily differentiate them from normal behavior.
* **Enable decision-makers** to quickly spot potential threats or spikes in network usage.
* **Support investigation and reporting** with graphical evidence of anomalies.

**🧱 Functional Components**

1. **Traffic Data Plotting**
   * Plots total bytes transferred across different dates or time windows.
   * Gives an overview of network load and traffic volume over time.
   * Helps identify patterns like peak usage hours, days, or weeks.
2. **Anomaly Highlighting**
   * Clearly marks the anomalous data points on the traffic graph using a contrasting color (commonly red).
   * Shows temporal position of anomalies in relation to regular activity.
   * Can help indicate whether anomalies are isolated incidents or part of a larger trend.
3. **Threshold Visualization (Optional)**
   * Displays the anomaly detection threshold (e.g., based on Isolation Forest scores).
   * Allows viewers to understand which data points exceed normal bounds.
   * Provides a visual representation of the AI model's decision boundary.
4. **Interactive Charts (if implemented)**
   * Users can hover over data points to view exact dates and values.
   * Filters for zooming in on specific time ranges.
   * Dropdowns to select different metrics or time granularities (e.g., daily, weekly, hourly).

**📊 Types of Visualizations Used**

| **Visualization Type** | **Purpose** |
| --- | --- |
| **Line Graph** | Shows traffic trends over time. Smooth and ideal for continuous data. |
| **Scatter Overlay** | Highlights detected anomalies by marking specific points on the line graph. |
| **Threshold Line** | A horizontal line representing the anomaly threshold (optional). Adds context. |
| **Zoom-in Section (optional)** | Allows deep dives into a specific time frame to examine anomalies in detail. |

**⚙️ Features and Customizations**

* **Custom Styling**: Colors, font sizes, and legends can be tailored for better readability.
* **Responsiveness**: Adapts to different screen sizes and formats for accessibility.
* **Modular Integration**: Easily integrates with upstream detection and downstream reporting modules.
* **Export Capability (optional)**: Allows exporting charts as images or PDFs for presentations and reports.

**📁 Output and Deliverables**

* **Final Graphical Output**:
  + One or more graphs saved as .png, .jpg, or rendered inline in a dashboard or report.
  + Anomalies clearly labeled for easy interpretation.
* **User Interpretation Layer**:
  + Graphs include titles, axis labels, legends, and tooltips to guide understanding.
  + Each graph is accompanied by a short summary describing the observations.

**✅ Benefits of the Visualization Module**

| **Benefit** | **Description** |
| --- | --- |
| **Improved Clarity** | Raw anomaly scores become instantly understandable through visuals. |
| **Faster Decision-Making** | Graphs reduce the time needed to identify threats. |
| **Enhanced Communication** | Easy to share with teams, managers, or clients. |
| **Audit and Forensics** | Visual logs can serve as forensic evidence of network anomalies. |

**📌 Use Case Scenarios**

* **Security Analysts**: Use visual trends to correlate anomalies with real-world events (e.g., scheduled maintenance, attacks).
* **Managers**: Gain high-level insight into system health without delving into technical logs.
* **Report Generation**: Automatically embed visuals into end-of-day or end-of-week security summaries.

**✅ Module 4: Interactive Charts**

**🎯 Objective**

The goal of the **Interactive Charts module** is to provide dynamic, user-friendly visualizations that allow stakeholders to dive deeper into network traffic patterns and anomalies. This module enhances user engagement by enabling real-time exploration of the data and detection results through interactive elements. It offers an intuitive way for users to interact with the visualizations, zooming into specific time periods, filtering data, or selecting particular metrics to analyze in detail.

**🧩 Key Features**

1. **Interactivity with Traffic Data**
   * Users can click, hover, or drag to interact with the data points in the visualization.
   * Zoom-in and zoom-out features allow users to focus on specific time frames or events.
   * Real-time data updates and interactions without refreshing the page.
2. **Anomaly Filtering**
   * Users can filter anomalies based on severity levels, time range, or traffic patterns.
   * Allows focusing on specific IP addresses or time periods with spikes or dips in activity.
3. **Customizable Charts**
   * Users can customize chart types (e.g., line, bar, scatter) to explore different perspectives of the data.
   * Users can select different metrics or features (e.g., total bytes, protocol types, or user locations) for deeper analysis.
4. **Dynamic Anomaly Scoring**
   * Display anomaly scores on the chart, allowing users to assess the magnitude of the detected anomaly.
   * Anomalies are clearly distinguished with color-coding (e.g., red for high severity, orange for moderate, and green for low).
5. **Time Granularity Control**
   * Allow users to adjust the time granularity (e.g., hourly, daily, weekly) to see patterns at different scales.
   * Enable comparative analysis by overlaying different time ranges on the same chart.

**📊 Types of Interactive Charts**

| **Chart Type** | **Purpose** |
| --- | --- |
| **Line Chart** | Ideal for showing trends over time, such as network traffic or anomaly detection over hours, days, or weeks. |
| **Bar Chart** | Shows the volume of traffic across different categories or groups, such as IP addresses, protocols, or ports. |
| **Scatter Plot** | Helps in identifying anomalies by plotting network data points and visually separating outliers. |
| **Histogram** | Displays the distribution of traffic volume, highlighting any unusual spikes or outliers. |
| **Heatmap** | Visualizes the intensity of traffic across time or between different regions or protocols. |

**⚙️ User Interaction Features**

1. **Hover Tooltips**
   * Display additional data details when the user hovers over a point in the chart.
   * Tooltips can show raw values, anomaly scores, IP addresses, or additional traffic statistics.
2. **Click-to-Drilldown**
   * Users can click on a data point or anomaly to drill down into a more detailed view.
   * Display further information, such as associated IP addresses, user-agent information, or session details.
3. **Dynamic Filtering**
   * Users can select from multiple filtering options to narrow down the displayed data, such as:
     + Time frame (e.g., last 24 hours, past week)
     + Traffic type (e.g., inbound vs. outbound)
     + Anomaly severity (e.g., high, medium, low)
   * Filters update the charts in real-time, enabling seamless exploration.
4. **Chart Customization**
   * Users can toggle between different chart types (line, bar, etc.).
   * Allow users to adjust chart settings, such as axis scales (logarithmic vs. linear), and color themes.

**🧠 Benefits of Interactive Charts**

| **Benefit** | **Description** |
| --- | --- |
| **Enhanced User Experience** | Interactive charts empower users to explore the data themselves, offering a personalized experience. |
| **Deeper Insights** | Enables users to analyze the data from various angles, uncovering hidden patterns and anomalies. |
| **Quick Decision-Making** | Real-time interactivity helps users rapidly make decisions based on up-to-date insights. |
| **Better Data Understanding** | Provides a more intuitive way to digest complex data and anomaly patterns. |

**📁 Output and Deliverables**

* **Interactive Dashboard**: The interactive charts are embedded within a Streamlit dashboard, allowing users to freely navigate and explore the data.
* **Exportable Charts**: Users can download customized charts as PNG or PDF for reporting purposes.
* **Real-time Updates**: The charts automatically refresh as new data is loaded or anomalies are detected, ensuring users always have access to the latest insights.

**✅ Benefits of Interactive Charts Module**

| **Benefit** | **Description** |
| --- | --- |
| **Increased Engagement** | Users are more likely to interact and understand the data when they can directly manipulate visual elements. |
| **Improved Insight Generation** | The ability to filter and zoom into data allows for better insights into trends and anomalies. |
| **Faster Problem Identification** | Interactive elements highlight critical areas of concern, making it easier to spot potential threats or issues. |
| **Collaboration** | Provides a tool for collaborative discussions, where multiple team members can analyze the same data and draw conclusions together. |

**📌 Use Case Scenarios**

* **Security Analysts**: Use interactive charts to analyze anomalies in real-time, zooming in on unusual traffic patterns or unexpected behavior.
* **Managers/Executives**: High-level insights into network traffic trends, allowing for quick decisions on resource allocation or security interventions.
* **Incident Response Teams**: Deep dive into anomalies to assess potential security breaches or operational issues.

**🔧 Technical Details**

* **Streamlit Integration**: The interactive charts are implemented using Plotly’s rich interactivity features and embedded into the Streamlit app.
* **Real-time Filtering**: Filters and interactivity are powered by JavaScript functions within the Streamlit app, ensuring responsiveness and smooth interaction.

**📦 Module 5: IP Lookup**

**Overview:**

The IP Lookup module allows users to investigate and analyze the network traffic data related to specific source IP addresses. By entering a source IP address, users can view the corresponding traffic details such as bytes sent, the destination IPs, and other relevant metrics. This functionality helps in identifying patterns, potential security issues, and overall network activity associated with individual IPs.

**Features:**

* **IP Address Input:** Users can input any source IP to query the traffic logs.
* **IP Traffic Data:** Displays rows of traffic data where the provided IP appears in the sIPs (source IP) column.
* **Traffic Analysis:**
  + Total bytes sent by the given IP.
  + Traffic distribution by destination IP addresses.
* **Real-Time Lookup:** The tool provides real-time results as users input the IP address.
* **Data Display:** The module shows a detailed table and summary statistics related to the specified IP.

**User Interface:**

* **Input Field:** A text input field allows users to type in the IP address they wish to investigate.
* **Data Output:**
  + Displays matching rows where the source IP (sIPs) matches the entered IP.
  + Shows the total bytes sent by the IP.
  + Displays a traffic distribution table with the breakdown of traffic by destination IP (rIPs).

**How It Works:**

1. **User Input:**
   * The user enters a source IP address in the input field.
2. **Data Filtering:**
   * The system searches through the dataset (usually a CSV or database) and filters rows where the source IP (sIPs) matches the entered IP.
3. **Result Display:**
   * If matching data is found, the corresponding rows are displayed in a table.
   * The module also calculates the total bytes sent by the source IP and presents it as a summary.
4. **Traffic Distribution:**
   * The module aggregates and shows the distribution of traffic based on destination IPs (rIPs) associated with the source IP.

**Code Implementation:**

**Parameters:**

* **IP Address (User Input):**
  + Type: String
  + Description: The source IP address provided by the user to investigate.
  + Example: 192.168.1.1

**Output:**

1. **Matching Traffic Data:** A table of rows where the source IP (sIPs) matches the input IP.
   * Columns include source IP, destination IP, bytes sent, etc.
2. **Total Bytes Sent:** A summary metric showing the total bytes sent by the input IP.
   * Example: Total Bytes Sent by IP 192.168.1.1: 1234567 bytes
3. **Traffic Distribution by Destination IP:** A table showing the breakdown of traffic for each destination IP.
   * Example:
   * Destination IP | Bytes Sent
   * --------------------|-------------
   * 192.168.1.2 | 234567
   * 10.0.0.1 | 456789

**Edge Cases and Error Handling:**

* **No Matches Found:** If the entered IP is not found in the dataset, a warning message is displayed: "No data found for IP: [IP address]."
* **Empty Input:** If the user does not input anything, a message prompts them to enter an IP address.

**Use Cases:**

* **Network Traffic Analysis:** Quickly view the network activity for specific source IP addresses.
* **Security Audits:** Investigate suspicious or unknown IPs to understand their behavior in the network.
* **Troubleshooting:** Identify abnormal traffic patterns or data volume associated with specific IP addresses.

**Future Enhancements:**

* **Geolocation Integration:** Integrate a geolocation API to show details like the country, city, and ISP of the IP address.
* **Visualization:** Include graphs (like bar charts or pie charts) to visualize the traffic distribution and byte usage.
* **Advanced Filtering:** Allow users to filter based on date ranges, protocols, or other traffic characteristics.

**Module 6: Graphical Flow Visualization**

**Objective:**

This module aims to provide a clear graphical representation of data flows within a network, focusing on IP relationships. It uses **Sankey diagrams** to depict the flow of data between source and destination IPs, and **network graphs** to show the interconnectivity between multiple IPs.

**Key Features:**

* **Sankey Diagrams:** Provides a flowchart-style visualization showing how much data is transferred between source and destination IPs.
* **Network Graphs:** Visualizes how different IPs are connected, showcasing the relationships and interactions between them.
* **Interactivity:** Allows users to explore connections in more detail, helping to identify high-traffic paths or unusual patterns.

**Use Cases:**

* **Traffic Analysis:** Helps identify the volume and direction of traffic between different IPs, which can be useful in understanding network bottlenecks, peak usage times, or abnormal traffic spikes.
* **Network Monitoring:** Visualizes the structure of the network and the flow of data, making it easier to detect unusual connections or unauthorized IP interactions.

**Benefits:**

* **Improved Network Understanding:** Provides a comprehensive, visual representation of IP interactions.
* **Anomaly Detection:** Helps identify unusual flows or relationships that might indicate security issues or network problems.

**Module 7: Rolling Anomaly Rate**

**Objective:**

The **Rolling Anomaly Rate** module is designed to visualize how the anomaly detection rate changes over time, allowing users to monitor patterns, trends, and the frequency of anomalies in network traffic. This helps in identifying periods of unusual activity and potential security concerns.

**Key Features:**

* **Time-Series Visualization:** Shows how the anomaly rate evolves over time, providing insight into when abnormal traffic spikes occur.
* **Rolling Average:** Smoothens out fluctuations in the anomaly detection process, helping to focus on broader trends instead of short-term anomalies.
* **Dynamic Updates:** Updates in real-time to reflect current anomaly detection rates, offering a continuous view of the network’s health.

**Use Cases:**

* **Trend Analysis:** Helps track long-term trends in network anomalies, identifying periods of high security risks or network instability.
* **Proactive Monitoring:** By observing rolling anomaly rates, teams can anticipate potential problems before they escalate, ensuring quicker response times.
* **Performance Optimization:** By tracking anomaly rates, network managers can optimize network performance by addressing issues that repeatedly trigger anomalies.

**Benefits:**

* **Early Detection of Issues:** Provides an early warning system for network problems or security threats based on detected anomalies.
* **Enhanced Network Health Monitoring:** Helps in maintaining a healthy network by identifying abnormal activity over extended periods.

**4. Conclusion**

This system provides a comprehensive, real-time view of network traffic, identifying and alerting on anomalies using machine learning. The interactive visualizations allow for detailed analysis, and the alert system ensures timely detection of potential network issues.

**Future Work**

* **Scalability**: Improve the system to handle larger datasets with more advanced algorithms (e.g., deep learning).
* **Integration with Security Tools**: Integrate with tools like IDS (Intrusion Detection Systems) for enhanced security features.
* **Model Improvement**: Experiment with other anomaly detection algorithms like **DBSCAN** or **Autoencoders** for better accuracy.

**Program:**

import streamlit as st

import pandas as pd

import plotly.express as px

import seaborn as sns

import matplotlib.pyplot as plt

import plotly.graph\_objects as go

from sklearn.ensemble import IsolationForest

from sklearn.svm import OneClassSVM

import streamlit as st

import pandas as pd

import numpy as np

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

from sklearn.ensemble import RandomForestClassifier

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

import plotly.express as px

import joblib

import io

import plotly.graph\_objects as go

from scipy.stats import gaussian\_kde

import networkx as nx

from pyvis.network import Network

st.set\_page\_config(page\_title="SOC Framework for Network Traffic Analysis and Threat Detection", layout="wide")

st.markdown(

"""

<style>

.stApp {

background-color: #0f1c2e;

color: white;

}

</style>

""",

unsafe\_allow\_html=True

)

# Load the dataset

@st.cache\_data(ttl=36)

def load\_data():

data = pd.read\_csv('network\_traffic.csv')

data['startDate'] = pd.to\_datetime(data['startDate'])

data['endDate'] = pd.to\_datetime(data['endDate'])

data['session\_duration'] = (data['endDate'] - data['startDate']).dt.total\_seconds()

data['packet\_to\_byte\_ratio'] = data['sPackets'] / data['sBytesSum']

return data

st.title("Network Traffic Analysis & Threat Detection")

# 📌 \*Streamlit App with Tabs\*

tab1, tab2 = st.tabs(["⚡ Monitoring Dashboard", "🛡 Cybersecurity & Attack Analysis"])

with tab1:

df = load\_data()

st.subheader("Sample Data")

st.dataframe(df.head())

with st.container():

col1, col2 = st.columns(2, gap="small")

col3, col4 = st.columns(2, gap="small")

# Chart 1: Histogram (Session Duration Distribution)

with col1:

fig1 = px.histogram(df, x="session\_duration", nbins=50,

title="Session Duration Distribution",

labels={"session\_duration": "Duration (Seconds)"})

fig1.update\_xaxes(tickmode="linear")

st.plotly\_chart(fig1, use\_container\_width=True)

# Chart 2: Scatter Plot (Bytes Sent vs. Packets Sent)

with col2:

fig2 = px.scatter(df, x="sPackets", y="sBytesSum",

title="Bytes Sent vs. Packets Sent",

labels={"sPackets": "Packets Sent", "sBytesSum": "Bytes Sent"})

st.plotly\_chart(fig2, use\_container\_width=True)

# Chart 3: Line Chart (Bytes Sent Over Time)

with col3:

df['Timestamp'] = pd.to\_datetime(df['startDate']) # Assuming startDate is in your dataset

fig3 = px.line(df, x="Timestamp", y="sBytesSum",

title="Total Bytes Sent Over Time",

labels={"sBytesSum": "Bytes Sent", "Timestamp": "Time"})

fig3.update\_yaxes(range=[df["sBytesSum"].min(), df["sBytesSum"].max()])

st.plotly\_chart(fig3, use\_container\_width=True)

# Chart 4: Bar Chart (Protocol Distribution)

with col4:

df\_protocol = df['protocol'].value\_counts().reset\_index()

df\_protocol.columns = ['Protocol', 'Count']

fig4 = px.bar(

df\_protocol,

x="Protocol", y="Count",

title="Protocol Distribution",

labels={"Protocol": "Protocol", "Count": "Count"}

)

st.plotly\_chart(fig4, use\_container\_width=True)

# Chart 5: Line Chart (Inter-packet Delay vs. Time)

with col1:

fig5 = px.line(df, x="Timestamp", y="sInterPacketAvg",

title="Inter-packet Delay Over Time",

labels={"sInterPacketAvg": "Avg Inter-packet Delay (ms)", "Timestamp": "Time"})

fig5.update\_yaxes(range=[df["sInterPacketAvg"].min(), df["sInterPacketAvg"].max()])

st.plotly\_chart(fig5, use\_container\_width=True)

# Chart 6: Forecast Error (If applicable)

with col2:

df["Forecast Error"] = df["sBytesSum"] - df["sBytesSum"].shift(1)

df.dropna(subset=["Forecast Error"], inplace=True)

fig6 = px.scatter(df, x="Timestamp", y="Forecast Error",

title="Forecast Error Over Time",

labels={"Forecast Error": "Error (Bytes)", "Timestamp": "Time"},

color\_discrete\_sequence=["#ff7f0e"])

st.plotly\_chart(fig6, use\_container\_width=True)

# Chart 7: Traffic Session Length Distribution

with col3:

fig7 = px.histogram(df, x="session\_duration", nbins=50,

title="Traffic Session Length Distribution",

labels={"session\_duration": "Session Length (Seconds)"})

st.plotly\_chart(fig7, use\_container\_width=True)

# Chart 8: Total Traffic (Sent Bytes vs. Received Bytes)

with col4:

# Step 1: Group and sum

total\_traffic = df.groupby("Timestamp")[["sBytesSum", "rBytesSum"]].sum().reset\_index()

# Step 2: Melt to long form

total\_traffic\_long = total\_traffic.melt(id\_vars=["Timestamp"],

value\_vars=["sBytesSum", "rBytesSum"],

var\_name="Type", value\_name="Bytes")

# Step 3: Plot

fig8 = px.bar(

total\_traffic\_long,

x="Timestamp", y="Bytes",

color="Type",

title="Total Sent vs. Received Bytes",

labels={"Timestamp": "Time", "Bytes": "Bytes"},

barmode="group",

color\_discrete\_map={"sBytesSum": "#fa7a5f", "rBytesSum": "#89fa5f"}

)

st.plotly\_chart(fig8, use\_container\_width=True)

with st.container():

col1, col2 = st.columns(2, gap="small")

col3, col4 = st.columns(2, gap="small")

# Chart 1: Line Chart (Packets Received Over Time)

with col1:

fig1 = px.line(df, x="Timestamp", y="rPackets",

title="Packets Received Over Time",

labels={"rPackets": "Packets Received", "Timestamp": "Time"})

fig1.update\_yaxes(range=[df["rPackets"].min(), df["rPackets"].max()])

st.plotly\_chart(fig1, use\_container\_width=True)

# Chart 2: Bar Chart (Traffic by Source IP)

with col2:

ip\_traffic = df['sIPs'].value\_counts().reset\_index().head(10)

ip\_traffic.columns = ['Source IP', 'Traffic Volume']

fig2 = px.bar(

ip\_traffic,

x="Source IP", y="Traffic Volume",

title="Top 10 Source IPs by Traffic Volume",

labels={"Source IP": "Source IP", "Traffic Volume": "Traffic Volume"}

)

st.plotly\_chart(fig2, use\_container\_width=True)

# Chart 3: Heatmap (Correlation Between Traffic Variables)

with col3:

correlation\_matrix = df[["sBytesSum", "rBytesSum", "sPackets", "rPackets", "session\_duration"]].corr()

fig3 = px.imshow(correlation\_matrix, text\_auto=True,

title="Correlation Between Traffic Variables")

st.plotly\_chart(fig3, use\_container\_width=True)

# Chart 4: Line Chart (Received Bytes vs. Session Duration)

with col4:

fig4 = px.line(df, x="session\_duration", y="rBytesSum",

title="Received Bytes vs. Session Duration",

labels={"rBytesSum": "Bytes Received", "session\_duration": "Session Duration"})

fig4.update\_yaxes(range=[df["rBytesSum"].min(), df["rBytesSum"].max()])

st.plotly\_chart(fig4, use\_container\_width=True)

# Chart 5: Bar Chart (Top 10 Destination IPs by Traffic)

with col1:

dest\_ip\_traffic = df['rIPs'].value\_counts().reset\_index().head(10)

dest\_ip\_traffic.columns = ['Destination IP', 'Traffic Volume']

fig5 = px.bar(

dest\_ip\_traffic,

x="Destination IP", y="Traffic Volume",

title="Top 10 Destination IPs by Traffic Volume",

labels={"Destination IP": "Destination IP", "Traffic Volume": "Traffic Volume"}

)

st.plotly\_chart(fig5, use\_container\_width=True)

# Chart 6: Line Chart (Inter-packet Delay vs. Traffic Volume)

with col2:

fig6 = px.line(df, x="sInterPacketAvg", y="sBytesSum",

title="Inter-packet Delay vs. Traffic Volume",

labels={"sInterPacketAvg": "Avg Inter-packet Delay (ms)", "sBytesSum": "Bytes Sent"})

fig6.update\_yaxes(range=[df["sBytesSum"].min(), df["sBytesSum"].max()])

st.plotly\_chart(fig6, use\_container\_width=True)

# Chart 7: Bar Chart (Session Duration by Source IP)

with col3:

session\_duration\_ip = (

df.groupby('sIPs')['session\_duration']

.mean()

.reset\_index()

.sort\_values('session\_duration', ascending=False)

.head(10)

)

# Optional: Rename to make things super clear

session\_duration\_ip.columns = ['Source IP', 'Average Session Duration (Seconds)']

fig7 = px.bar(

session\_duration\_ip,

x="Source IP", y="Average Session Duration (Seconds)",

title="Top 10 Source IPs by Average Session Duration",

labels={

"Source IP": "Source IP",

"Average Session Duration (Seconds)": "Average Session Duration (Seconds)"

}

)

st.plotly\_chart(fig7, use\_container\_width=True)

# Chart 8: Histogram (Session Duration vs. Packets Sent)

with col4:

fig8 = px.histogram(df, x="session\_duration", y="sPackets",

title="Session Duration vs. Packets Sent",

labels={"session\_duration": "Session Duration (Seconds)", "sPackets": "Packets Sent"})

st.plotly\_chart(fig8, use\_container\_width=True)

with st.container():

col1, col2 = st.columns(2, gap="small")

col3, col4 = st.columns(2, gap="small")

# Chart 1: Top 10 Source MAC Addresses (sMACs)

with col1:

smac\_traffic = df['sMACs'].value\_counts().reset\_index().head(10)

smac\_traffic.columns = ['Source MAC Address', 'Traffic Volume']

fig1 = px.bar(

smac\_traffic,

x="Source MAC Address", y="Traffic Volume",

title="Top 10 Source MAC Addresses by Traffic Volume",

labels={

"Source MAC Address": "Source MAC Address",

"Traffic Volume": "Traffic Volume"

}

)

st.plotly\_chart(fig1, use\_container\_width=True)

# Chart 2: Top 10 Destination MAC Addresses (rMACs)

with col2:

rmac\_traffic = df['rMACs'].value\_counts().reset\_index().head(10)

rmac\_traffic.columns = ['Destination MAC Address', 'Traffic Volume']

fig2 = px.bar(

rmac\_traffic,

x="Destination MAC Address", y="Traffic Volume",

title="Top 10 Destination MAC Addresses by Traffic Volume",

labels={

"Destination MAC Address": "Destination MAC Address",

"Traffic Volume": "Traffic Volume"

}

)

st.plotly\_chart(fig2, use\_container\_width=True)

with st.container():

col1, col2 = st.columns(2, gap="small")

# Chart 1: Top 10 Protocols (Pie Chart)

with col1:

protocol\_traffic = df['protocol'].value\_counts().reset\_index().head(10)

protocol\_traffic.columns = ['protocol', 'count'] # Rename columns for clarity

fig1 = px.pie(protocol\_traffic, names="protocol", values="count",

title="Top 10 Protocols by Traffic Volume",

labels={"protocol": "Protocol", "count": "Traffic Volume"})

st.plotly\_chart(fig1, use\_container\_width=True)

# Chart 2: NST\_M\_Label (Pie Chart)

with col2:

nst\_m\_label\_traffic = df['NST\_M\_Label'].value\_counts().reset\_index()

nst\_m\_label\_traffic.columns = ['NST\_M\_Label', 'count'] # Rename columns

fig2 = px.pie(nst\_m\_label\_traffic, names="NST\_M\_Label", values="count",

title="NST\_M\_Label Distribution",

labels={"NST\_M\_Label": "NST\_M\_Label", "count": "Count"})

st.plotly\_chart(fig2, use\_container\_width=True)

df = df.dropna()

with st.container():

col1, col2 = st.columns(2, gap="small")

col3, col4 = st.columns(2, gap="small")

# Distribution: sPackets

with col1:

fig1 = px.histogram(df, x="sPackets", nbins=50, title="Distribution of sPackets",

labels={"sPackets": "sPackets"})

st.plotly\_chart(fig1, use\_container\_width=True)

# Distribution: rPackets

with col2:

fig2 = px.histogram(df, x="rPackets", nbins=50, title="Distribution of rPackets",

labels={"rPackets": "rPackets"})

st.plotly\_chart(fig2, use\_container\_width=True)

# Distribution: sBytesSum

with col3:

fig3 = px.histogram(df, x="sBytesSum", nbins=50, title="Distribution of sBytesSum",

labels={"sBytesSum": "sBytesSum"})

st.plotly\_chart(fig3, use\_container\_width=True)

# Distribution: rBytesSum

with col4:

fig4 = px.histogram(df, x="rBytesSum", nbins=50, title="Distribution of rBytesSum",

labels={"rBytesSum": "rBytesSum"})

st.plotly\_chart(fig4, use\_container\_width=True)

with st.container():

col5, col6 = st.columns(2, gap="small")

col7, col8 = st.columns(2, gap="small")

# Distribution: sBytesMax

with col5:

fig5 = px.histogram(df, x="sBytesMax", nbins=50, title="Distribution of sBytesMax",

labels={"sBytesMax": "sBytesMax"})

st.plotly\_chart(fig5, use\_container\_width=True)

# Distribution: rBytesMax

with col6:

fig6 = px.histogram(df, x="rBytesMax", nbins=50, title="Distribution of rBytesMax",

labels={"rBytesMax": "rBytesMax"})

st.plotly\_chart(fig6, use\_container\_width=True)

# Distribution: sBytesMin

with col7:

fig7 = px.histogram(df, x="sBytesMin", nbins=50, title="Distribution of sBytesMin",

labels={"sBytesMin": "sBytesMin"})

st.plotly\_chart(fig7, use\_container\_width=True)

# Distribution: rBytesMin

with col8:

fig8 = px.histogram(df, x="rBytesMin", nbins=50, title="Distribution of rBytesMin",

labels={"rBytesMin": "rBytesMin"})

st.plotly\_chart(fig8, use\_container\_width=True)

with st.container():

col9, col10 = st.columns(2, gap="small")

col11, col12 = st.columns(2, gap="small")

# Distribution: sBytesAvg

with col9:

fig9 = px.histogram(df, x="sBytesAvg", nbins=50, title="Distribution of sBytesAvg",

labels={"sBytesAvg": "sBytesAvg"})

st.plotly\_chart(fig9, use\_container\_width=True)

# Distribution: rBytesAvg

with col10:

fig10 = px.histogram(df, x="rBytesAvg", nbins=50, title="Distribution of rBytesAvg",

labels={"rBytesAvg": "rBytesAvg"})

st.plotly\_chart(fig10, use\_container\_width=True)

# Distribution: sLoad

with col11:

fig11 = px.histogram(df, x="sLoad", nbins=50, title="Distribution of sLoad",

labels={"sLoad": "sLoad"})

st.plotly\_chart(fig11, use\_container\_width=True)

# Distribution: rLoad

with col12:

fig12 = px.histogram(df, x="rLoad", nbins=50, title="Distribution of rLoad",

labels={"rLoad": "rLoad"})

st.plotly\_chart(fig12, use\_container\_width=True)

with st.container():

col13, col14 = st.columns(2, gap="small")

col15, col16 = st.columns(2, gap="small")

# Distribution: sPayloadSum

with col13:

fig13 = px.histogram(df, x="sPayloadSum", nbins=50, title="Distribution of sPayloadSum",

labels={"sPayloadSum": "sPayloadSum"})

st.plotly\_chart(fig13, use\_container\_width=True)

# Distribution: rPayloadSum

with col14:

fig14 = px.histogram(df, x="rPayloadSum", nbins=50, title="Distribution of rPayloadSum",

labels={"rPayloadSum": "rPayloadSum"})

st.plotly\_chart(fig14, use\_container\_width=True)

# Distribution: sPayloadMax

with col15:

fig15 = px.histogram(df, x="sPayloadMax", nbins=50, title="Distribution of sPayloadMax",

labels={"sPayloadMax": "sPayloadMax"})

st.plotly\_chart(fig15, use\_container\_width=True)

# Distribution: rPayloadMax

with col16:

fig16 = px.histogram(df, x="rPayloadMax", nbins=50, title="Distribution of rPayloadMax",

labels={"rPayloadMax": "rPayloadMax"})

st.plotly\_chart(fig16, use\_container\_width=True)

with st.container():

col17, col18 = st.columns(2, gap="small")

col19, col20 = st.columns(2, gap="small")

# Distribution: sPayloadMin

with col17:

fig17 = px.histogram(df, x="sPayloadMin", nbins=50, title="Distribution of sPayloadMin",

labels={"sPayloadMin": "sPayloadMin"})

st.plotly\_chart(fig17, use\_container\_width=True)

# Distribution: rPayloadMin

with col18:

fig18 = px.histogram(df, x="rPayloadMin", nbins=50, title="Distribution of rPayloadMin",

labels={"rPayloadMin": "rPayloadMin"})

st.plotly\_chart(fig18, use\_container\_width=True)

# Distribution: sPayloadAvg

with col19:

fig19 = px.histogram(df, x="sPayloadAvg", nbins=50, title="Distribution of sPayloadAvg",

labels={"sPayloadAvg": "sPayloadAvg"})

st.plotly\_chart(fig19, use\_container\_width=True)

# Distribution: rPayloadAvg

with col20:

fig20 = px.histogram(df, x="rPayloadAvg", nbins=50, title="Distribution of rPayloadAvg",

labels={"rPayloadAvg": "rPayloadAvg"})

st.plotly\_chart(fig20, use\_container\_width=True)

with st.container():

col21, col22 = st.columns(2, gap="small")

# Distribution: sInterPacketAvg

with col21:

fig21 = px.histogram(df, x="sInterPacketAvg", nbins=50, title="Distribution of sInterPacketAvg",

labels={"sInterPacketAvg": "sInterPacketAvg"})

st.plotly\_chart(fig21, use\_container\_width=True)

# Distribution: rInterPacketAvg

with col22:

fig22 = px.histogram(df, x="rInterPacketAvg", nbins=50, title="Distribution of rInterPacketAvg",

labels={"rInterPacketAvg": "rInterPacketAvg"})

st.plotly\_chart(fig22, use\_container\_width=True)

with tab2:

df = load\_data()

# Ensure the Timestamp is in datetime format

df['startDate'] = pd.to\_datetime(df['startDate'])

df['endDate'] = pd.to\_datetime(df['endDate'])

# Preprocess the data: Handle missing values

df.fillna(0, inplace=True)

# Feature selection for anomaly detection

X = df[['sPackets', 'rPackets', 'sBytesSum', 'rBytesSum', 'sLoad', 'rLoad']]

# Dynamic slider for Anomaly Contamination

anomaly\_contamination = st.slider(

"🤖 Set Anomaly Contamination Rate (Isolation Forest)",

0.001, 0.05, 0.01, step=0.005

)

# Train Isolation Forest model for anomaly detection with dynamic contamination rate

iso\_forest = IsolationForest(contamination=anomaly\_contamination)

df['anomaly\_iso'] = iso\_forest.fit\_predict(X)

anomalies\_iso = df[df['anomaly\_iso'] == -1]

# 1. Real-time Anomaly Count & Alert System

st.write(f"\*Total Anomalies Detected:\* {len(anomalies\_iso) }")

if len(anomalies\_iso) > 1500:

st.warning("High number of anomalies detected!")

else:

st.success("Anomalies count is within expected range.")

fig\_iso = px.scatter(

anomalies\_iso, x='startDate', y='sBytesSum', color='sIPs',

title="Anomalies (Isolation Forest)"

)

st.plotly\_chart(fig\_iso, use\_container\_width=True)

# 2. Anomaly Trend Over Time (Time Series Plot)

fig\_time\_iso = px.line(

anomalies\_iso, x='startDate', y='sBytesSum', color='sIPs',

title="Anomalies Trend (Isolation Forest)"

)

st.plotly\_chart(fig\_time\_iso, use\_container\_width=True)

# 4. Correlation Matrix of Features

corr\_matrix = X.corr()

fig\_corr = px.imshow(corr\_matrix, title="Feature Correlation Matrix")

st.plotly\_chart(fig\_corr, use\_container\_width=True)

# 6. Boxplot for Outlier Detection

fig\_box = px.box(df, y="sBytesSum", title="Boxplot for sBytesSum", points='all')

st.plotly\_chart(fig\_box, use\_container\_width=True)

# 8. Feature Threshold Alerts (Manual Inspection)

threshold = 10000

high\_sBytes = df[df['sBytesSum'] > threshold]

if not high\_sBytes.empty:

st.warning(f"Found {len(high\_sBytes)} entries with sBytesSum greater than {threshold}")

# 9. Advanced Alerts (Email/Push Notification - Placeholder)

# Uncomment and implement below if you want to send an email in case of high anomaly count

def send\_alert\_email():

try:

with smtplib.SMTP('smtp.gmail.com', 587) as server:

server.starttls()

server.login("your\_email@gmail.com", "your\_password")

message = "Subject: Anomaly Alert\n\nHigh anomaly count detected."

server.sendmail("your\_email@gmail.com", "recipient\_email@gmail.com", message)

except Exception as e:

st.error(f"Error sending email: {e}")

if len(anomalies\_iso) > 2000:

send\_alert\_email()

# 10. Anomaly Summary (Top Anomalies)

st.write("\*Top 5 Anomalies Detected (by Isolation Forest):\*")

st.dataframe(anomalies\_iso[['sIPs', 'sBytesSum', 'startDate']].head())

with st.container():

# Time-of-Day and Day-of-Week Heatmap

st.markdown("### 🔥 Anomaly Heatmap by Time & Day")

anomalies\_iso['hour'] = anomalies\_iso['startDate'].dt.hour

anomalies\_iso['weekday'] = anomalies\_iso['startDate'].dt.day\_name()

heatmap\_data = anomalies\_iso.groupby(['weekday', 'hour']).size().unstack().fillna(0)

st.dataframe(heatmap\_data)

# Top Talkers Bar Chart

st.markdown("### 💡 Top Talkers by Average Sent Bytes")

top\_senders = df.groupby('sIPs')['sBytesSum'].mean().sort\_values(ascending=False).head(10)

st.bar\_chart(top\_senders)

st.markdown("### 📉 Feature Distribution: Anomalies vs Normal")

# Prepare data

anomaly\_data = df[df['anomaly\_iso'] == -1]['sBytesSum']

normal\_data = df[df['anomaly\_iso'] == 1]['sBytesSum']

# Generate KDE manually using scipy

x\_vals = np.linspace(min(df['sBytesSum']), max(df['sBytesSum']), 100)

kde\_anomaly = gaussian\_kde(anomaly\_data)(x\_vals)

kde\_normal = gaussian\_kde(normal\_data)(x\_vals)

# Create Plotly figure

fig\_dist = go.Figure()

fig\_dist.add\_trace(go.Scatter(

x=x\_vals, y=kde\_anomaly, fill='tozeroy', mode='lines',

name='Anomalies', line=dict(color='red')

))

fig\_dist.add\_trace(go.Scatter(

x=x\_vals, y=kde\_normal, fill='tozeroy', mode='lines',

name='Normal', line=dict(color='green')

))

fig\_dist.update\_layout(

title="sBytesSum Distribution: Anomalies vs Normal",

xaxis\_title="sBytesSum",

yaxis\_title="Density",

legend\_title="Legend",

template="plotly\_dark"

)

st.plotly\_chart(fig\_dist, use\_container\_width=True)

# Rolling Window Anomaly Rate

st.markdown("### ⏱ Rolling Anomaly Rate (1 Hour)")

df['is\_anomaly'] = (df['anomaly\_iso'] == -1).astype(int)

df\_sorted = df.sort\_values("startDate")

df\_sorted['anomaly\_rate\_rolling'] = (

df\_sorted.set\_index('startDate')['is\_anomaly']

.rolling('1H')

.mean()

.reset\_index(drop=True)

)

fig\_roll = px.line(df\_sorted, x='startDate', y='anomaly\_rate\_rolling', title="Rolling Anomaly Rate (1 Hour)")

st.plotly\_chart(fig\_roll, use\_container\_width=True)

# Auto Severity Score

st.markdown("### 🚨 Auto Severity Classification for Anomalies")

def calculate\_severity(row):

if row['sBytesSum'] > 20000 or row['sLoad'] > 0.8:

return "High"

elif row['sBytesSum'] > 10000:

return "Medium"

else:

return "Low"

anomalies\_iso['severity'] = anomalies\_iso.apply(calculate\_severity, axis=1)

st.dataframe(anomalies\_iso[['sIPs', 'sBytesSum', 'sLoad', 'severity']].sort\_values(by='severity', ascending=False))

# IP Lookup Tool

st.markdown("### 🕵 Investigate Specific Source IP")

ip\_input = st.text\_input("Enter Source IP to Inspect:")

if ip\_input:

st.dataframe(df[df['sIPs'] == ip\_input])

with st.container():

st.markdown("### 🧭 Radar Chart: Feature Profile")

# Get all numeric columns except the label

numeric\_cols = df.select\_dtypes(include='number').columns.tolist()

candidate\_features = [col for col in numeric\_cols if col != 'anomaly\_iso']

if len(candidate\_features) >= 2:

from sklearn.preprocessing import MinMaxScaler

import plotly.graph\_objects as go

radar\_df = df[candidate\_features + ['anomaly\_iso']].dropna().copy()

scaler = MinMaxScaler()

radar\_scaled = scaler.fit\_transform(radar\_df[candidate\_features])

radar\_scaled\_df = pd.DataFrame(radar\_scaled, columns=candidate\_features)

# Choose one anomaly and one normal

anomaly = radar\_scaled\_df[df['anomaly\_iso'] == -1].head(1)

normal = radar\_scaled\_df[df['anomaly\_iso'] == 1].head(1)

if not anomaly.empty and not normal.empty:

radar\_plot\_df = pd.concat([anomaly, normal])

labels = list(radar\_plot\_df.columns)

values = radar\_plot\_df.values

fig = go.Figure()

fig.add\_trace(go.Scatterpolar(r=values[0], theta=labels, fill='toself', name='Anomaly'))

fig.add\_trace(go.Scatterpolar(r=values[1], theta=labels, fill='toself', name='Normal'))

fig.update\_layout(polar=dict(radialaxis=dict(visible=True)), showlegend=True)

st.plotly\_chart(fig)

else:

st.info("Not enough anomaly or normal data for Radar Chart.")

else:

st.info("Not enough numeric features for Radar Chart.")

with st.container():

st.markdown("### 🌐 Sankey Diagram: Traffic Flow")

# Minimal working example — aggregate traffic flows

sankey\_data = df.groupby(['sIPs', 'rIPs'])['sBytesSum'].sum().reset\_index()

if len(sankey\_data) > 0:

all\_nodes = list(set(sankey\_data['sIPs']) | set(sankey\_data['rIPs']))

node\_map = {ip: i for i, ip in enumerate(all\_nodes)}

sources = sankey\_data['sIPs'].map(node\_map)

targets = sankey\_data['rIPs'].map(node\_map)

values = sankey\_data['sBytesSum']

fig = go.Figure(data=[go.Sankey(

node=dict(label=all\_nodes),

link=dict(source=sources, target=targets, value=values)

)])

st.plotly\_chart(fig)

else:

st.info("No data available to create Sankey Diagram.")

# -------------------------------------

edge\_attr = 'weight' if 'weight' in df.columns else None

# Create a container for the network graph

with st.container():

st.markdown("### 🕸 Interactive Network Graph")

# Create a network graph using NetworkX

if edge\_attr:

G = nx.from\_pandas\_edgelist(df, source='sIPs', target='rIPs', edge\_attr=edge\_attr)

else:

G = nx.from\_pandas\_edgelist(df, source='sIPs', target='rIPs')

# Initialize Pyvis Network for interactive graph

net = Network(height="600px", width="100%", notebook=True)

net.from\_nx(G)

# Customize the layout and appearance if needed

net.set\_options("""

var options = {

"physics": {

"enabled": true,

"barnesHut": {

"gravitationalConstant": -8000,

"springLength": 150

}

},

"nodes": {

"color": {

"background": "skyblue",

"border": "gray"

},

"size": 20

},

"edges": {

"color": "gray",

"width": 1

}

}

""")

# Generate the interactive HTML file for the network graph

net.show("network\_graph.html")

# Render the graph in the Streamlit app

HtmlFile = open("network\_graph.html", 'r', encoding='utf-8')

st.components.v1.html(HtmlFile.read(), height=600)