Pcap -> csv

ML PROJECT

* **Pcap -> csv**

The ISOT dataset is the combination of several existing publicly available malicious and non-malicious datasets. It contains the network traffic data in the form of pcap files

First need to extract required fields from pcap to csv format. Luckily wireshark has a cmd line tool called **tshark**. It even has a tool called **editcap.** I used it to convert the 10gb pcap file into smaller 17 files. I wrote a simple python script to convert the files into csv format

The following fields extracted from pcap files:

1. frame.number
2. frame.time\_epoch
3. frame.len
4. Ip.src
5. Ip.dst
6. Ip.proto
7. Tcp.srcport
8. Tcp.dstport
9. Tcp.flags
10. Tcp.window\_size
11. udp.srcport
12. udp.dstport
13. udp.length
14. dns.qry.name
15. http.host

**Script:**

import subprocess

pcap\_folder = r"C:\Users\Skeletron\Desktop\temp"

output\_folder = r"C:\Users\Skeletron\Desktop\dataset"

fields = [

"frame.number", "frame.time\_epoch", "frame.len",

"ip.src", "ip.dst", "ip.proto",

"tcp.srcport", "tcp.dstport", "tcp.flags", "tcp.window\_size",

"udp.srcport", "udp.dstport", "udp.length",

"dns.qry.name", "http.host"

]

fields\_arg = " ".join([f"-e {field}" for field in fields])

for i in range(1, 18): # Loop from 1 to 17

pcap\_file = f"{pcap\_folder}\\{i}.pcap"

csv\_file = f"{output\_folder}\\{i}.csv"

tshark\_cmd = f'tshark -r "{pcap\_file}" -T fields {fields\_arg} -E header=y -E separator=, -E quote=d > "{csv\_file}"'

subprocess.run(tshark\_cmd, shell=True, check=True)

print(f" Converted: {pcap\_file} -> {csv\_file}")

print("All files converted!")

* Also Adding label column using the dataset description provided

import pandas as pd

import os

# Malicious IPs from ISOT dataset

MALICIOUS\_IPS = {

'172.16.2.11', '172.16.0.2', '172.16.0.11',

'172.16.0.12', '172.16.2.12'

}

# Folder containing your CSV files

INPUT\_FOLDER = r"C:\Users\Skeletron\Desktop\dataset"

for filename in os.listdir(INPUT\_FOLDER):

if filename.endswith('.csv'):

file\_path = os.path.join(INPUT\_FOLDER, filename)

df = pd.read\_csv(file\_path)

if 'is\_malicious' not in df.columns:

df['is\_malicious'] = (

df['ip.src'].isin(MALICIOUS\_IPS) |

df['ip.dst'].isin(MALICIOUS\_IPS)

).astype(int)

# Overwrite original file

df.to\_csv(file\_path, index=False)

print(f" Added labels to {filename}")

Data Cleaning

* Data Cleaning

Making sure all the fields have the correct data types before working on them

Converting to appropriate datatypes

| Field | Before | After |
| --- | --- | --- |
| frame.number | Int64 | Int64 |
| frame.time\_epoch | float64 | datatime |
| frame.len | Int64 | Int64 |
| ip.src | object | category |
| ip.dst | object | category |
| ip.proto | object | Int64 |
| tcp.srcport | float64 | Int64 |
| tcp.dstport | float64 | Int64 |
| tcp.flags | object | Int64 |
| tcp.window\_size | float64 | Int64 |
| udp.srcport | float64 | Int64 |
| udp.dstport | float64 | Int64 |
| udp.length | float64 | Int64 |
| dns.qry.name | object | Int64 |
| http.host | object | Int64 |

Converting frame.time\_epoch to datetime

df['frame.time\_epoch'] = pd.to\_datetime(df['frame.time\_epoch'], unit='s', errors='coerce')

Handling multiple ips and converting to category type

df['ip.src'] = df['ip.src'].str.split(',').str[0]

df['ip.dst'] = df['ip.dst'].str.split(',').str[0]

df['ip.src'] = df['ip.src'].astype('category').cat.add\_categories('<empty>').fillna('<empty>')

df['ip.dst'] = df['ip.dst'].astype('category').cat.add\_categories('<empty>').fillna('<empty>')

Converting tcp.flag to int

def is\_hex(value):

try:

int(value, 16)

return True

except (ValueError, TypeError):

return False

df['tcp.flags'] = df['tcp.flags'].apply(lambda x: int(x, 16) if pd.notna(x) and is\_hex(x) else pd.NA).astype('Int64')

Converting rest of the numeric cols to int

numeric\_cols = [

'frame.number', 'frame.len', 'ip.proto', 'tcp.srcport',

'tcp.dstport', 'tcp.flags', 'tcp.window\_size',

'udp.srcport', 'udp.dstport', 'udp.length'

]

df[numeric\_cols] = df[numeric\_cols].apply(pd.to\_numeric, errors='coerce').astype('Int64')

Data Visualization (Before)

* **Data Visualization (Before)**

1. Missing Values

#missing values percentage

missing\_percent = df.isnull().mean() \* 100

plt.figure(figsize=(12, 6))

missing\_percent.plot(kind='bar', color='skyblue')

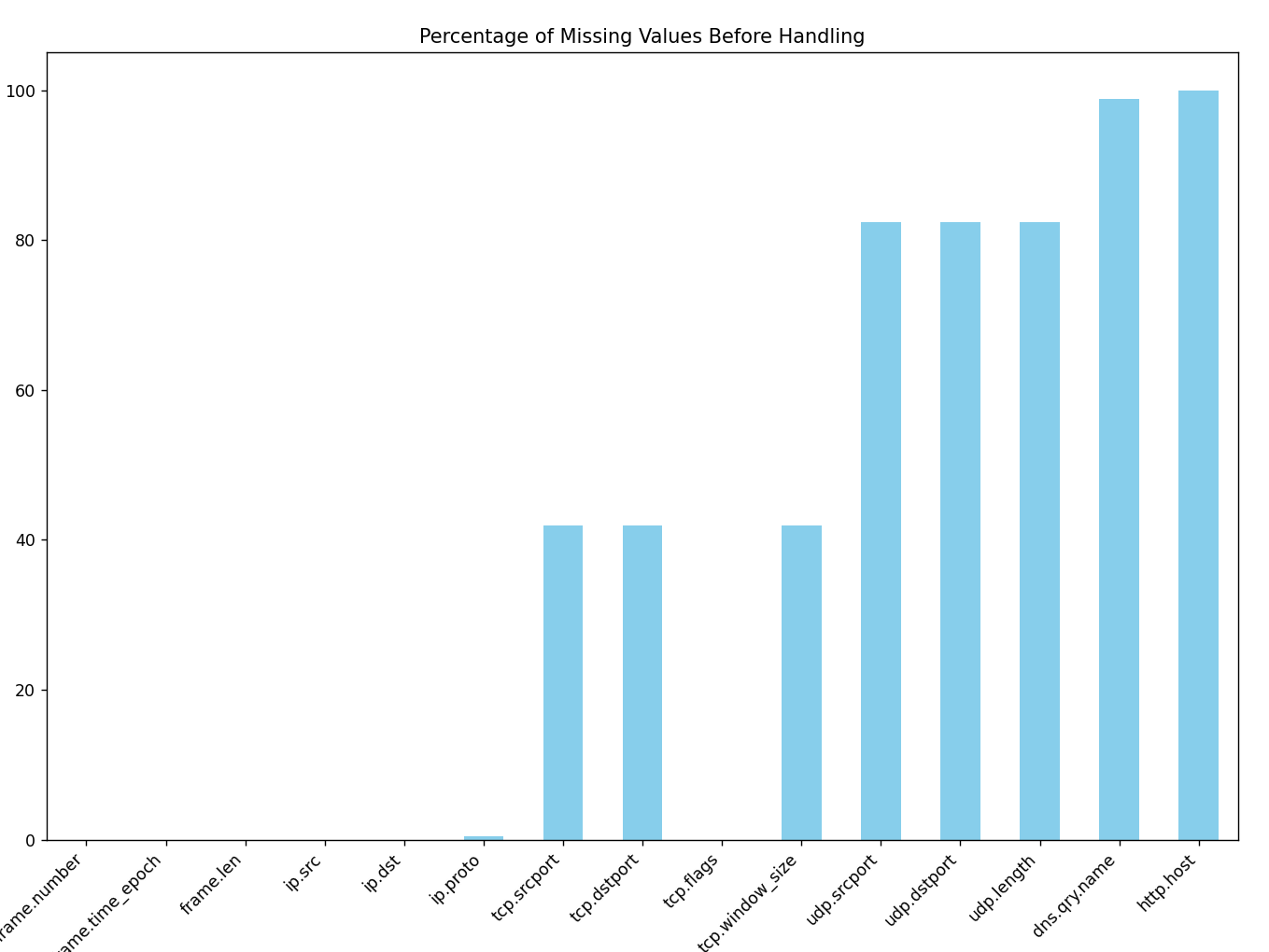
plt.title(f'Percentage of Missing Values Before Handling - {filename}')

plt.xlabel('Columns')

plt.ylabel('Percentage Missing (%)')

plt.xticks(rotation=45, ha="right")

plt.show()



1. Traffic Distribution

# Traffic Distribution

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

protocol\_counts = df['ip.proto'].value\_counts().rename({6: 'TCP', 17: 'UDP', 1: 'ICMP', 50: 'ESP', 103: 'PIM'})

protocol\_counts.plot(kind='bar', color='lightblue')

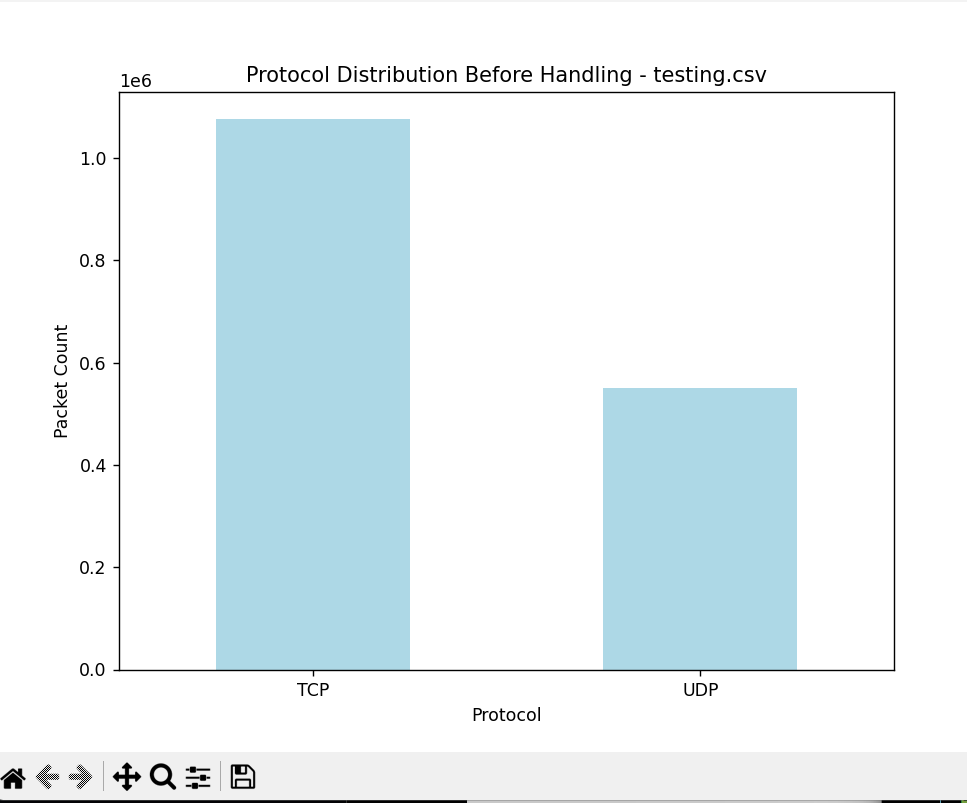
plt.title(f'Protocol Distribution Before Handling - {filename}')

plt.xlabel('Protocol')

plt.ylabel('Packet Count')

plt.xticks(rotation=0)

plt.show()



Handling

* Handling

Filling Missing and dropping dns.qry and http.host

import pandas as pd

import os

input\_folder = r"C:\Users\Skeletron\Desktop\dataset"

for filename in os.listdir(input\_folder):

if filename.endswith(".csv"):

file\_path = os.path.join(input\_folder, filename)

print(f"Processing file: {filename}")

df = pd.read\_csv(file\_path, low\_memory=False)

# Fillign missing values

df['tcp.window\_size'] = df['tcp.window\_size'].fillna(0)

df['tcp.flags'] = df['tcp.flags'].fillna(0)

df['tcp.srcport'] = df['tcp.srcport'].fillna(0)

df['tcp.dstport'] = df['tcp.dstport'].fillna(0)

df['udp.srcport'] = df['udp.srcport'].fillna(0)

df['udp.dstport'] = df['udp.dstport'].fillna(0)

df['udp.length'] = df['udp.length'].fillna(0)

df['ip.proto'] = df['ip.proto'].fillna(0)

# Dropping dns.qry.name and http.host due to high missing values

df.drop(columns=['dns.qry.name', 'http.host'], inplace=True)

df.to\_csv(file\_path, index=False)

print(f"Data handling complete for {filename} and saved.")

print("All CSVs handled successfully.")

Data Visualization(After)

* Data Visualization(After)

1. Missing Values

# Plot missing values percentage

missing\_percent = df.isnull().mean() \* 100

plt.figure(figsize=(12, 6))

missing\_percent.plot(kind='bar', color='skyblue')

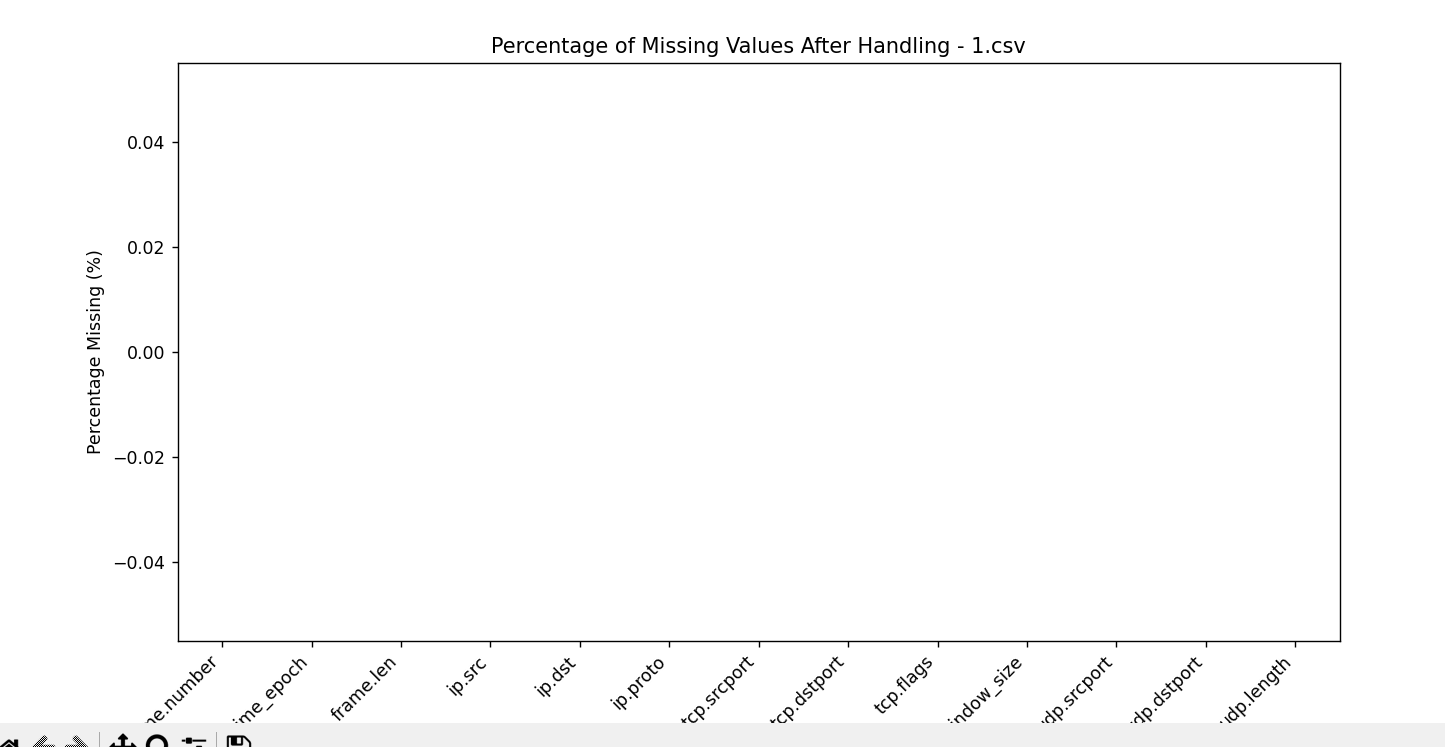
plt.title(f'Percentage of Missing Values After Handling - {filename}')

plt.xlabel('Columns')

plt.ylabel('Percentage Missing (%)')

plt.xticks(rotation=45, ha="right")

plt.show()



1. TCP flag distribution

# TCP/UDP Flag Distribution

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

tcp\_flag\_counts = df['tcp.flags'].value\_counts().sort\_index()

tcp\_flag\_counts.plot(kind='bar', color='lightcoral')

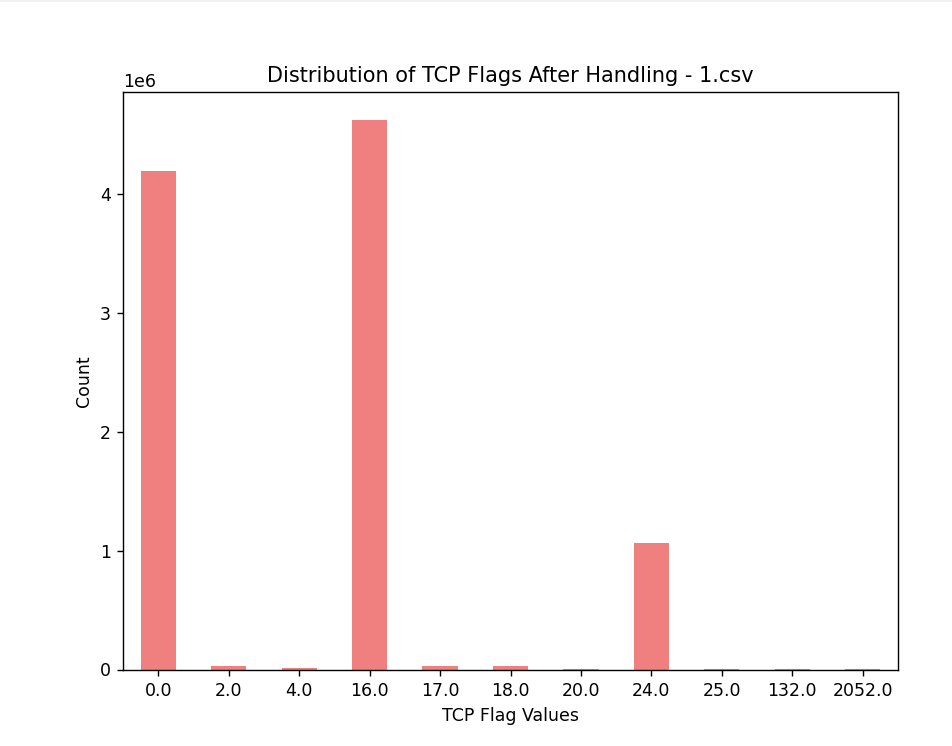
plt.title('Distribution of TCP Flags After Handling - testing.csv')

plt.xlabel('TCP Flag Values')

plt.ylabel('Count')

plt.xticks(rotation=0)

plt.show()



1. Traffic Distribution

# Protocol Distribution Check

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

protocol\_counts = df['ip.proto'].value\_counts().rename({6: 'TCP', 17: 'UDP', 1: 'ICMP', 50: 'ESP', 103: 'PIM'})

protocol\_counts.plot(kind='bar', color='lightblue')

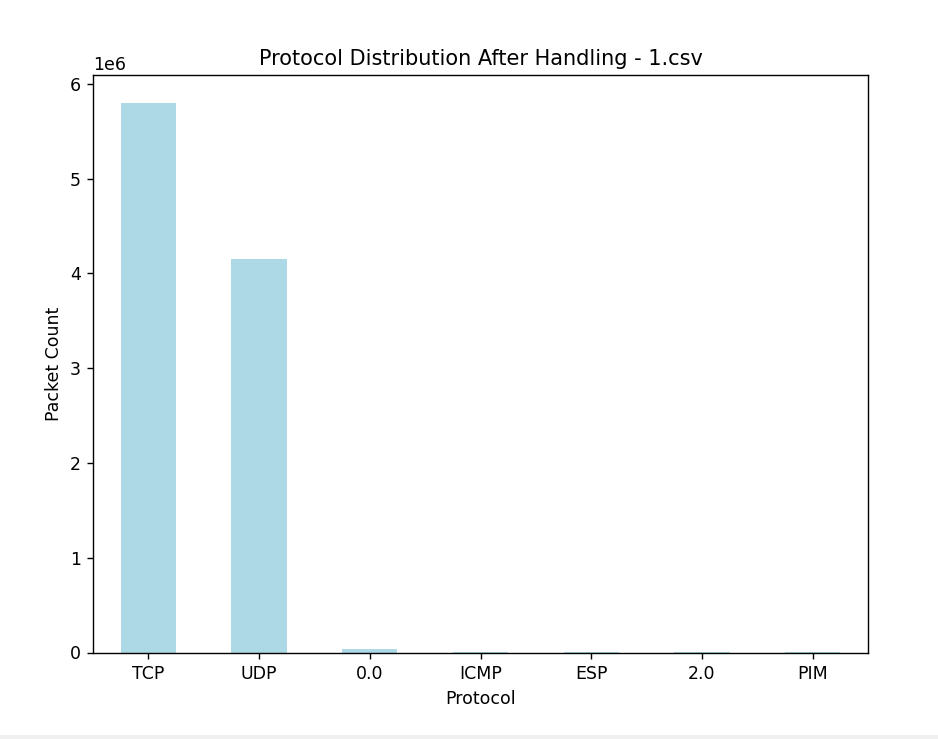
plt.title('Protocol Distribution After Handling - testing.csv')

plt.xlabel('Protocol')

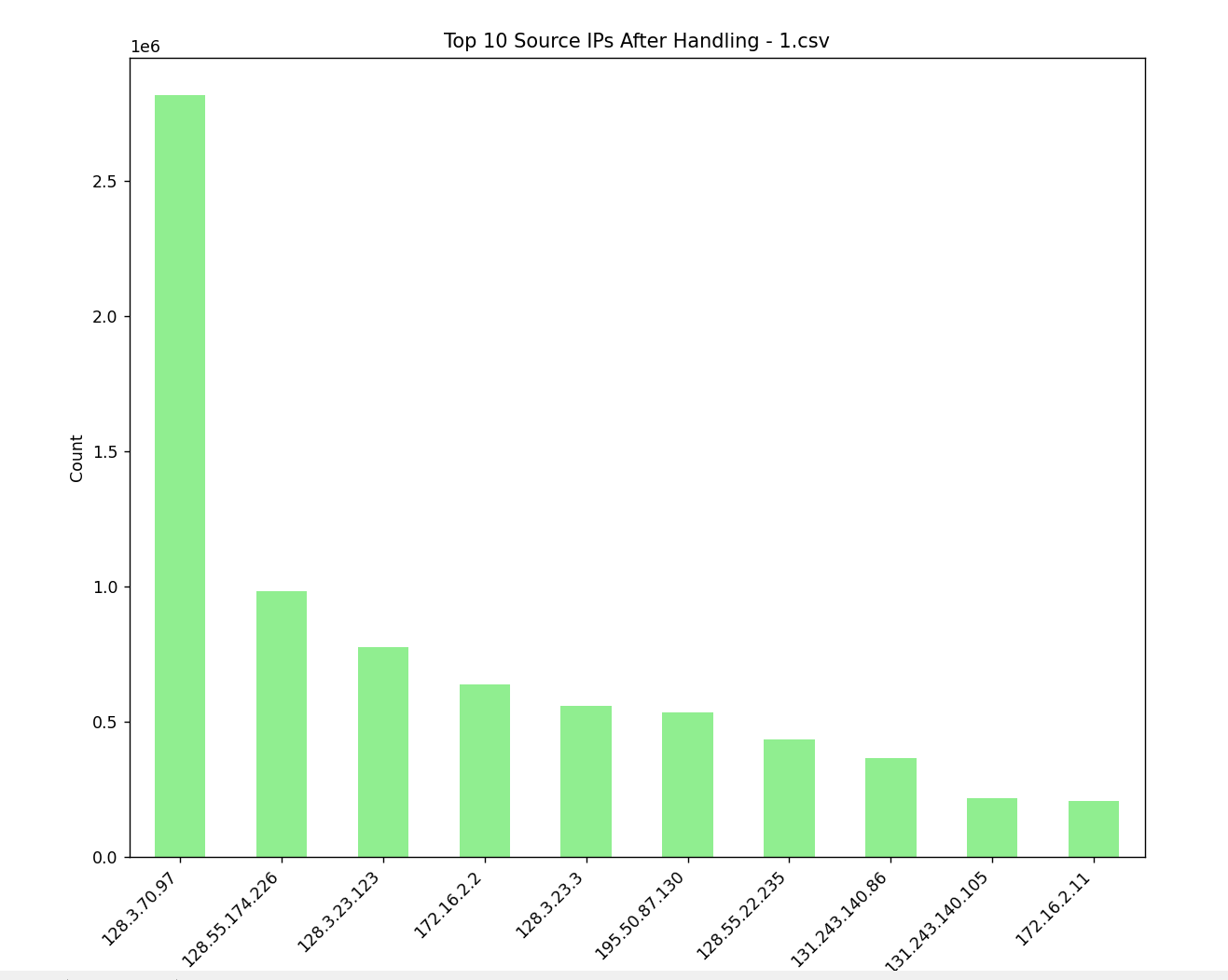
plt.ylabel('Packet Count')

plt.xticks(rotation=0)

plt.show()



1. Top 10 ips (src)
2. plt.figure(figsize=(10, 5))
3. df['ip.src'].value\_counts().head(10).plot(kind='bar', color='lightgreen')
4. plt.title('Top 10 Source IPs After Handling - testing.csv')
5. plt.xlabel('Source IP')
6. plt.ylabel('Count')
7. plt.xticks(rotation=45, ha="right")
8. plt.show()



5) Top 10 ips (dst)

plt.figure(figsize=(10, 5))

df['ip.dst'].value\_counts().head(10).plot(kind='bar', color='lightblue')

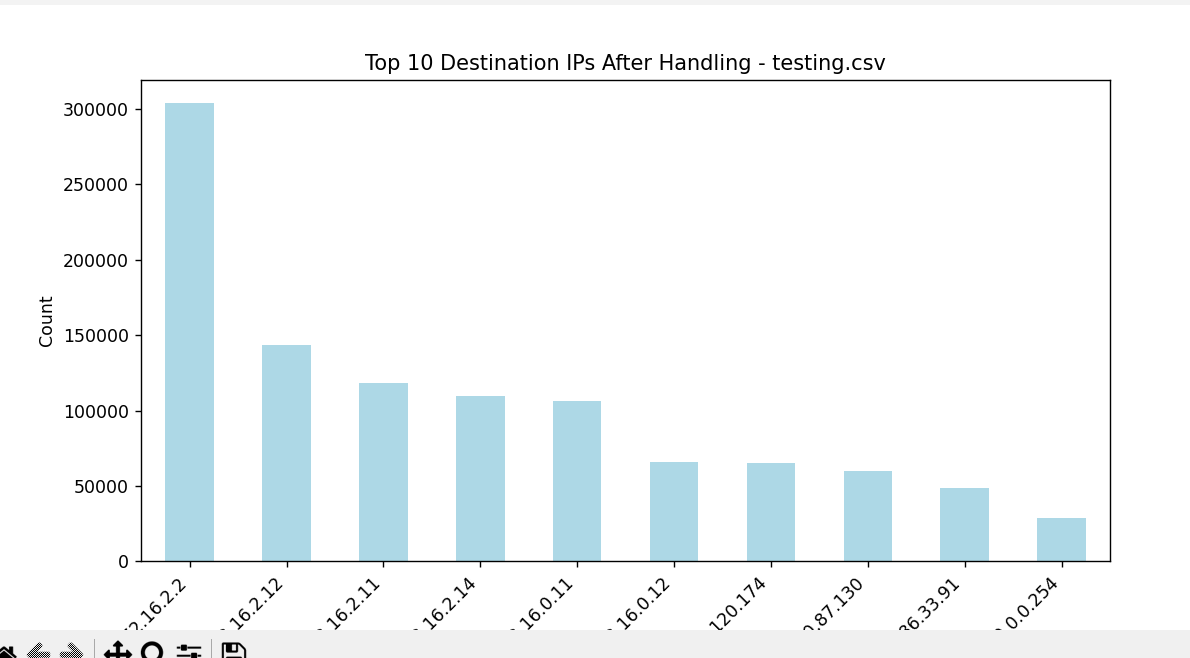
plt.title('Top 10 Destination IPs After Handling - testing.csv')

plt.xlabel('Destination IP')

plt.ylabel('Count')

plt.xticks(rotation=45, ha="right")

plt.show()



6) Malicious V/s Non Malicious Network

#Malicious vs Non-Malicious

plt.figure(figsize=(8, 5))

counts = df['is\_malicious'].value\_counts()

counts.index = counts.index.map({0: 'Non-Malicious', 1: 'Malicious'})

counts.plot(kind='bar', color=['green', 'red'], alpha=0.7)

plt.title('Malicious vs Non-Malicious Traffic Count')

plt.xlabel('Traffic Type')

plt.ylabel('Number of Flows')

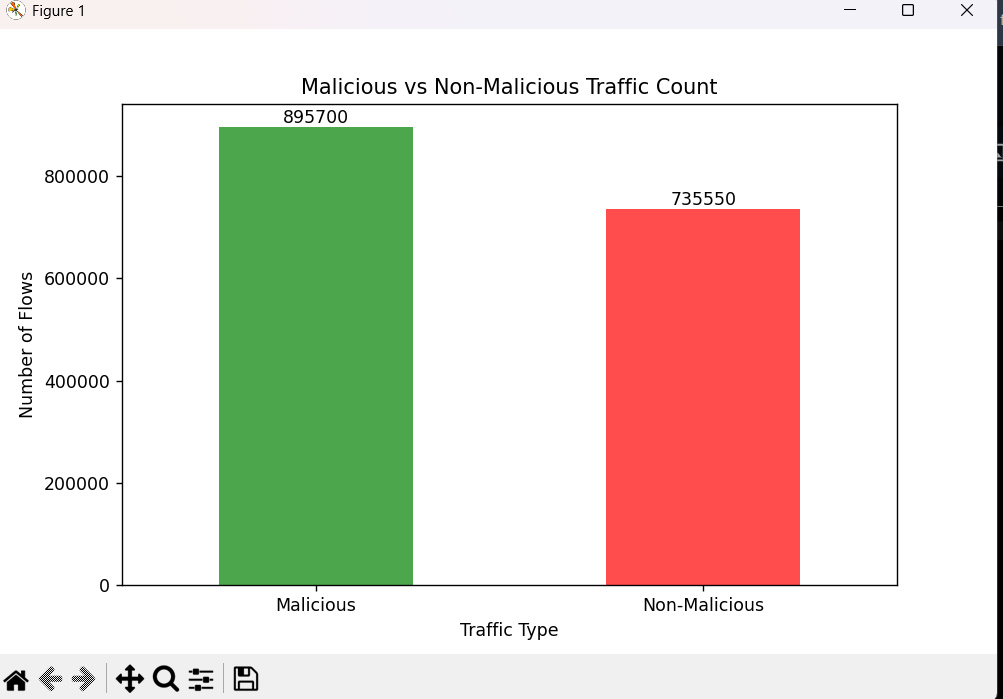
plt.xticks(rotation=0)

# Add exact counts on bars

for i, v in enumerate(counts):

plt.text(i, v, str(v), ha='center', va='bottom')

plt.show()



7)Correlation Table

plt.figure(figsize=(10, 8))

numeric\_df = df.select\_dtypes(include=['float64', 'int64'])

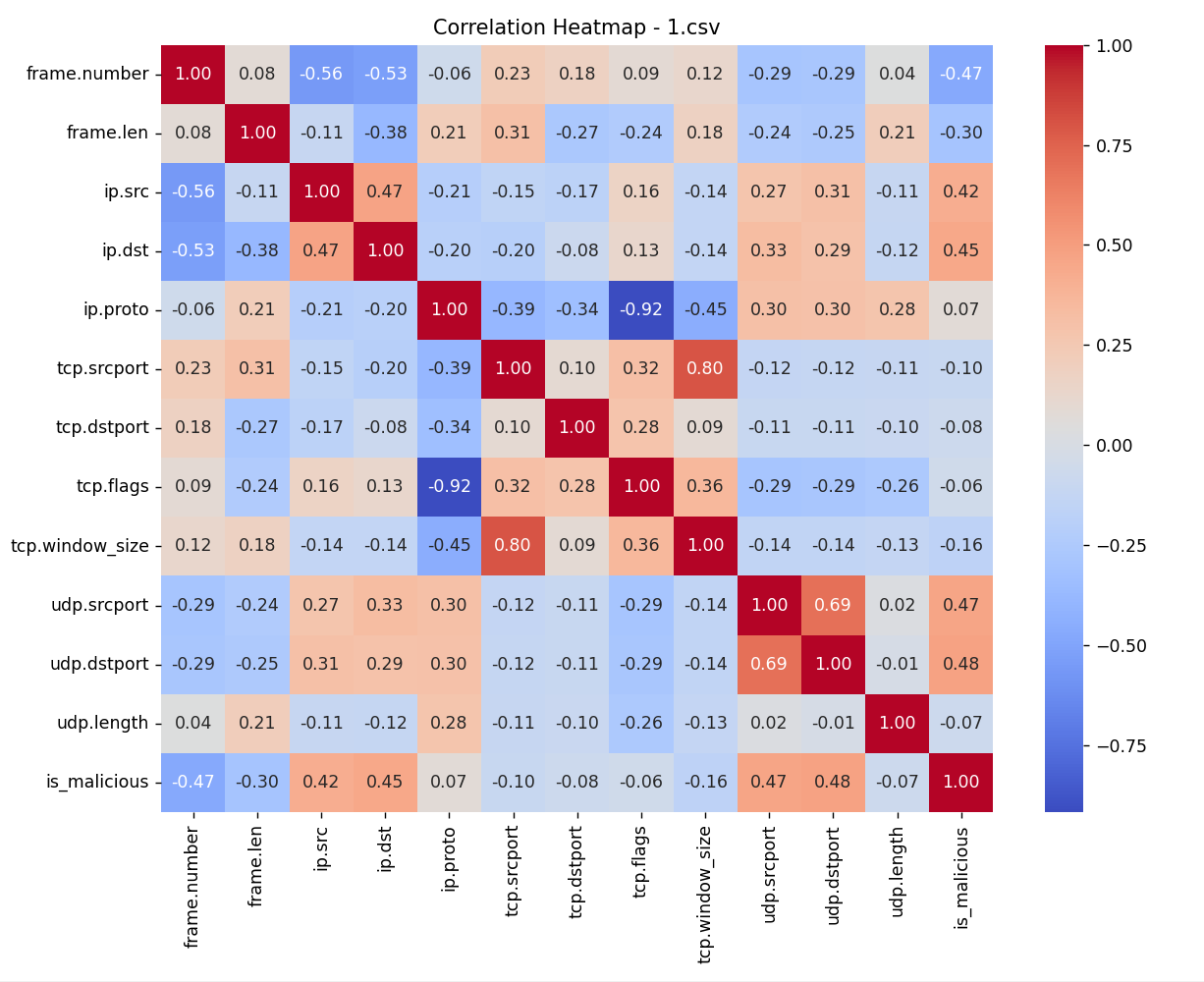
corr = numeric\_df.corr()

sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm')

plt.title('Correlation Heatmap - 1.csv')

plt.tight\_layout()

plt.show()



8)Frame Length Distribution

# Frame Length Distribution

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

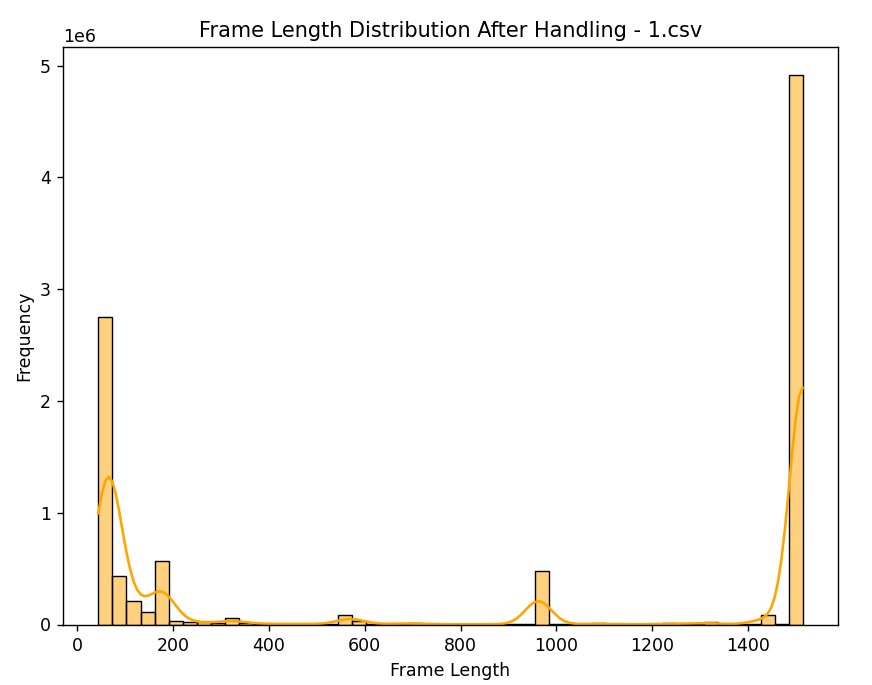
sns.histplot(df['frame.len'].dropna(), bins=50, kde=True, color='orange')

plt.title('Frame Length Distribution After Handling - testing.csv')

plt.xlabel('Frame Length')

plt.ylabel('Frequency')

plt.show()



Std/Normalization and Outliers

Std/Normalization and Outliers

Std/Normalization

* Most of our features (e.g., tcp.srcport, ip.proto, tcp.flags) are categorical **or** identifiers, not continuous.

* Tree-based models (like Random Tree, Decision Tree) which will be used in this project does not require scaling for performance.

Outlier Handing

* In cybersecurity/traffic analysis, outliers (e.g., extremely large frame.len or port anomalies) may indicate malicious activity.
* Removing these would reduce model effectiveness and suppress important threat indicators.
* In real-world traffic, such "irregular" values are expected and meaningful. Preserving them keeps the dataset true to the real environment
* Again Tree-Based Model are not affected by outliers.

Encoding

* Encoding

Ips are very crucial but in the current state cannot be processed. So, using label encoding to map unique ips to a int value

import pandas as pd

from sklearn.preprocessing import LabelEncoder

import joblib

import os

input\_folder = r"C:\Users\Skeletron\Desktop\dataset"

# Initialize LabelEncoders for ip.src and ip.dst

src\_encoder = LabelEncoder()

dst\_encoder = LabelEncoder()

# Collecting unique IPs from all CSVs

all\_src\_ips, all\_dst\_ips = set(), set()

# Scan through all CSVs to gather unique IPs

for filename in os.listdir(input\_folder):

if filename.endswith(".csv"):

file\_path = os.path.join(input\_folder, filename)

df = pd.read\_csv(file\_path, low\_memory=False)

# Add unique IPs to the sets

all\_src\_ips.update(df['ip.src'].dropna().unique())

all\_dst\_ips.update(df['ip.dst'].dropna().unique())

# Fit the encoders on the combined unique IPs

src\_encoder.fit(list(all\_src\_ips))

dst\_encoder.fit(list(all\_dst\_ips))

# Save the encoders for future use

joblib.dump(src\_encoder, os.path.join(input\_folder, 'ip\_src\_encoder.pkl'))

joblib.dump(dst\_encoder, os.path.join(input\_folder, 'ip\_dst\_encoder.pkl'))

print("Encoders trained and saved successfully.")

# Encode IPs in each CSV

for filename in os.listdir(input\_folder):

if filename.endswith(".csv"):

file\_path = os.path.join(input\_folder, filename)

df = pd.read\_csv(file\_path, low\_memory=False)

# Apply Label Encoding

df['ip.src'] = src\_encoder.transform(df['ip.src'].fillna('<empty>'))

df['ip.dst'] = dst\_encoder.transform(df['ip.dst'].fillna('<empty>'))

# Save the encoded data (overwrite the original)

df.to\_csv(file\_path, index=False)

print(f"{filename} encoded successfully.")

print("All CSVs encoded successfully.")

Feature Extraction

### **Feature Extraction**

Raw packet-level data was aggregated into flow-based features to better capture traffic behavior. Each flow was identified using a combination of source/destination IPs, ports, and protocol. The following features were extracted:

* **Traffic Stats:** Packet count, total bytes, average and standard deviation of packet size.
* **Temporal Features:** Flow duration, bytes per second, packets per second.
* **Behavioral Indicators:** TCP flag count and number of small packets (<128 bytes).
* **Flow Labeling:** A flow was marked malicious if any packet within it was labeled as such.

Port fields were unified across TCP and UDP, and missing values (e.g., in standard deviation) were handled appropriately. These flow-level features allow machine learning models to detect botnet activity more effectively than raw packet data.

CODE:

import os

import pandas as pd

import numpy as np

def process\_file(path):

df = pd.read\_csv(path)

# Convert timestamp

df['frame.time\_epoch'] = pd.to\_datetime(

df['frame.time\_epoch'],

errors='coerce',

format='mixed'

)

# Unified ports

df['src\_port'] = df['tcp.srcport'].replace(0, np.nan).fillna(df['udp.srcport'])

df['dst\_port'] = df['tcp.dstport'].replace(0, np.nan).fillna(df['udp.dstport'])

# Flow ID

df['flow\_id'] = (

df['ip.src'].astype(str) + "-" +

df['ip.dst'].astype(str) + "-" +

df['ip.proto'].astype(str) + "-" +

df['src\_port'].astype(str) + "-" +

df['dst\_port'].astype(str)

).apply(hash).astype('int64')

# Aggregation

agg\_df = df.groupby('flow\_id').agg(

ip\_src=('ip.src', 'first'),

ip\_dst=('ip.dst', 'first'),

proto=('ip.proto', 'first'),

src\_port=('src\_port', 'first'),

dst\_port=('dst\_port', 'first'),

packet\_count=('frame.len', 'count'),

total\_bytes=('frame.len', 'sum'),

avg\_packet\_size=('frame.len', 'mean'),

std\_packet\_size=('frame.len', 'std'),

flow\_start=('frame.time\_epoch', 'min'),

flow\_end=('frame.time\_epoch', 'max'),

is\_malicious=('is\_malicious', 'max'),

tcp\_flag\_count=('tcp.flags', lambda x: x.notna().sum()),

small\_packets=('frame.len', lambda x: (x < 128).sum())

).reset\_index()

# Derived features

agg\_df['flow\_start'] = pd.to\_datetime(agg\_df['flow\_start'], errors='coerce')

agg\_df['flow\_end'] = pd.to\_datetime(agg\_df['flow\_end'], errors='coerce')

agg\_df['flow\_duration'] = (agg\_df['flow\_end'] - agg\_df['flow\_start']).dt.total\_seconds().replace(0, 1e-6)

agg\_df['bytes\_per\_second'] = agg\_df['total\_bytes'] / agg\_df['flow\_duration']

agg\_df['packets\_per\_second'] = agg\_df['packet\_count'] / agg\_df['flow\_duration']

# Cleanup

agg\_df['std\_packet\_size'] = agg\_df['std\_packet\_size'].fillna(-1)

agg\_df.drop(columns=['flow\_start', 'flow\_end'], inplace=True)

# Round

round\_cols = ['avg\_packet\_size', 'std\_packet\_size', 'flow\_duration', 'bytes\_per\_second', 'packets\_per\_second']

agg\_df[round\_cols] = agg\_df[round\_cols].round(6)

# Order columns

column\_order = [

'flow\_id', 'ip\_src', 'ip\_dst', 'proto', 'src\_port', 'dst\_port',

'packet\_count', 'total\_bytes', 'avg\_packet\_size', 'std\_packet\_size', 'small\_packets',

'flow\_duration', 'bytes\_per\_second', 'packets\_per\_second', 'tcp\_flag\_count', 'is\_malicious'

]

agg\_df = agg\_df[column\_order]

# Save

agg\_df.to\_csv(path, index=False)

print(f"Processed: {os.path.basename(path)}")

# Batch run on all 17 CSVs

base\_path = r"C:\Users\Skeletron\Desktop\dataset"

for i in range(1, 18):

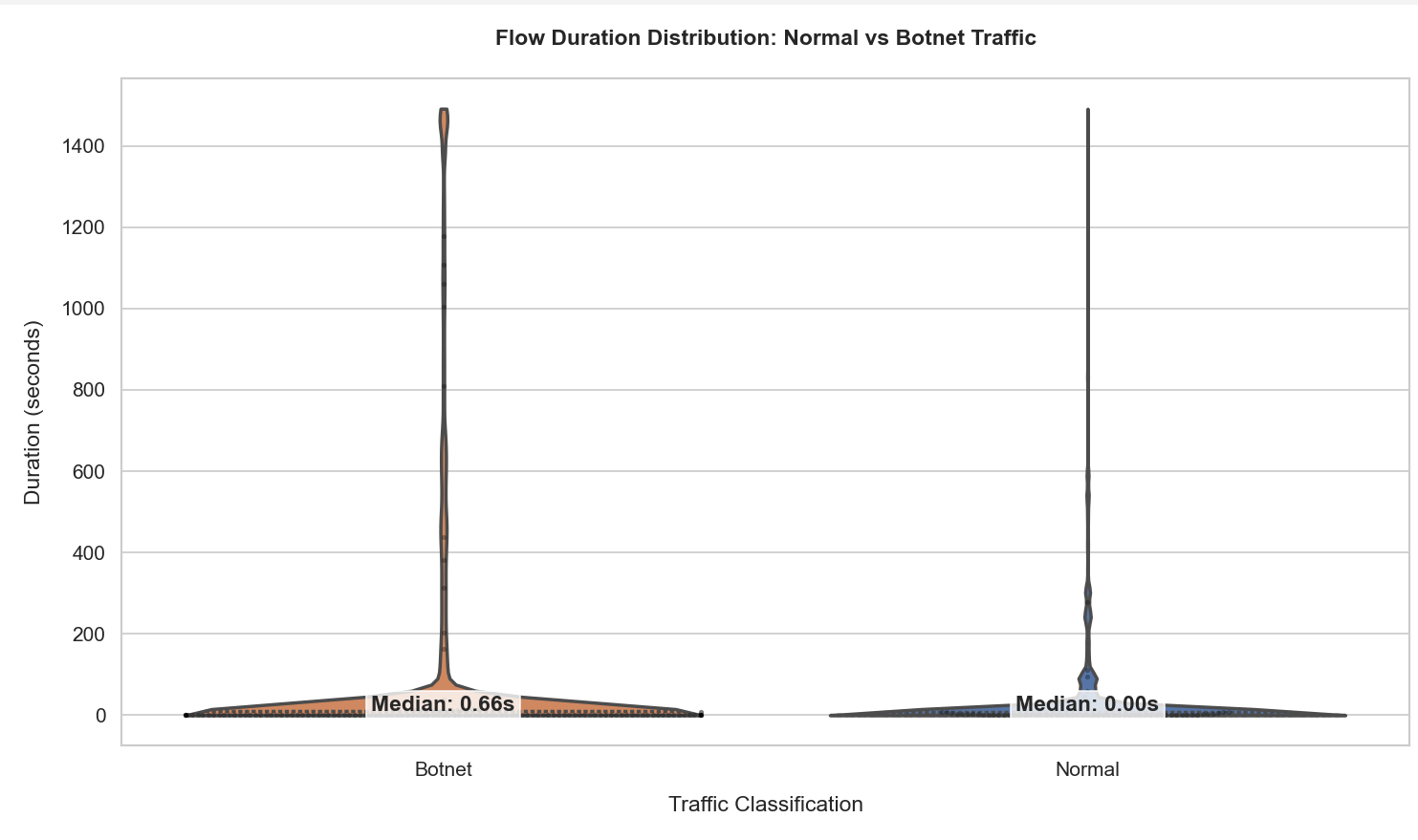
file\_path = os.path.join(base\_path, f"{i}.csv")

process\_file(file\_path)

Visualisation

**Visualisation**

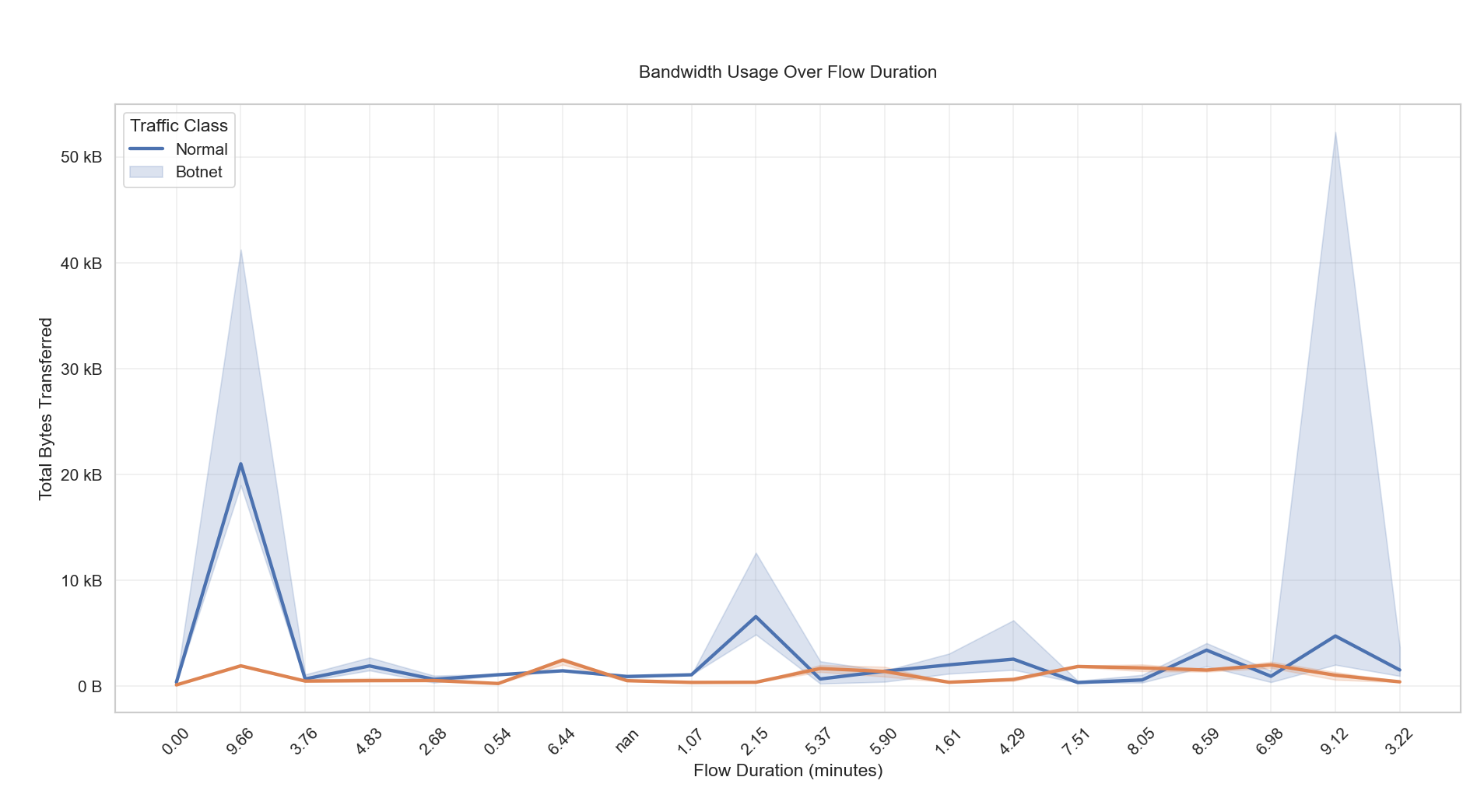
Violin Plot:



**Description:** This violin plot visualizes how long individual flows last in botnet vs normal traffic. The quartiles and swarm points highlight density and variance, while median annotations make the comparison clearer.

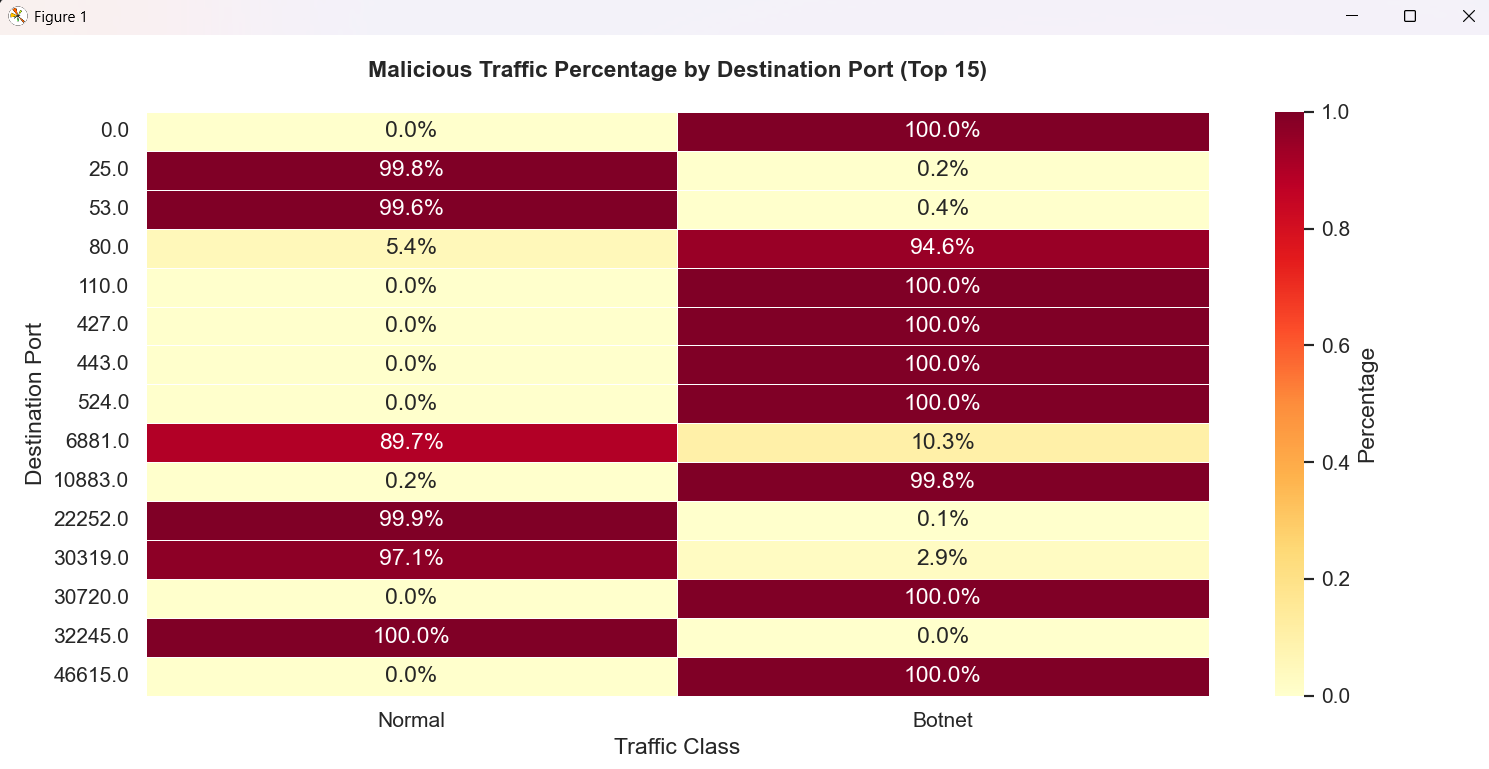
**Insight:** Botnet traffic often has distinct flow duration patterns, helping distinguish it from normal activity.

Traffic Timeline:

  
  
**Description:** Line plot showing the median bytes transferred across different flow duration bins (in minutes), split by traffic type. Error bars show 95% confidence intervals.

**Insight:** Reveals how bandwidth usage patterns change over time and differ between normal and malicious traffic.

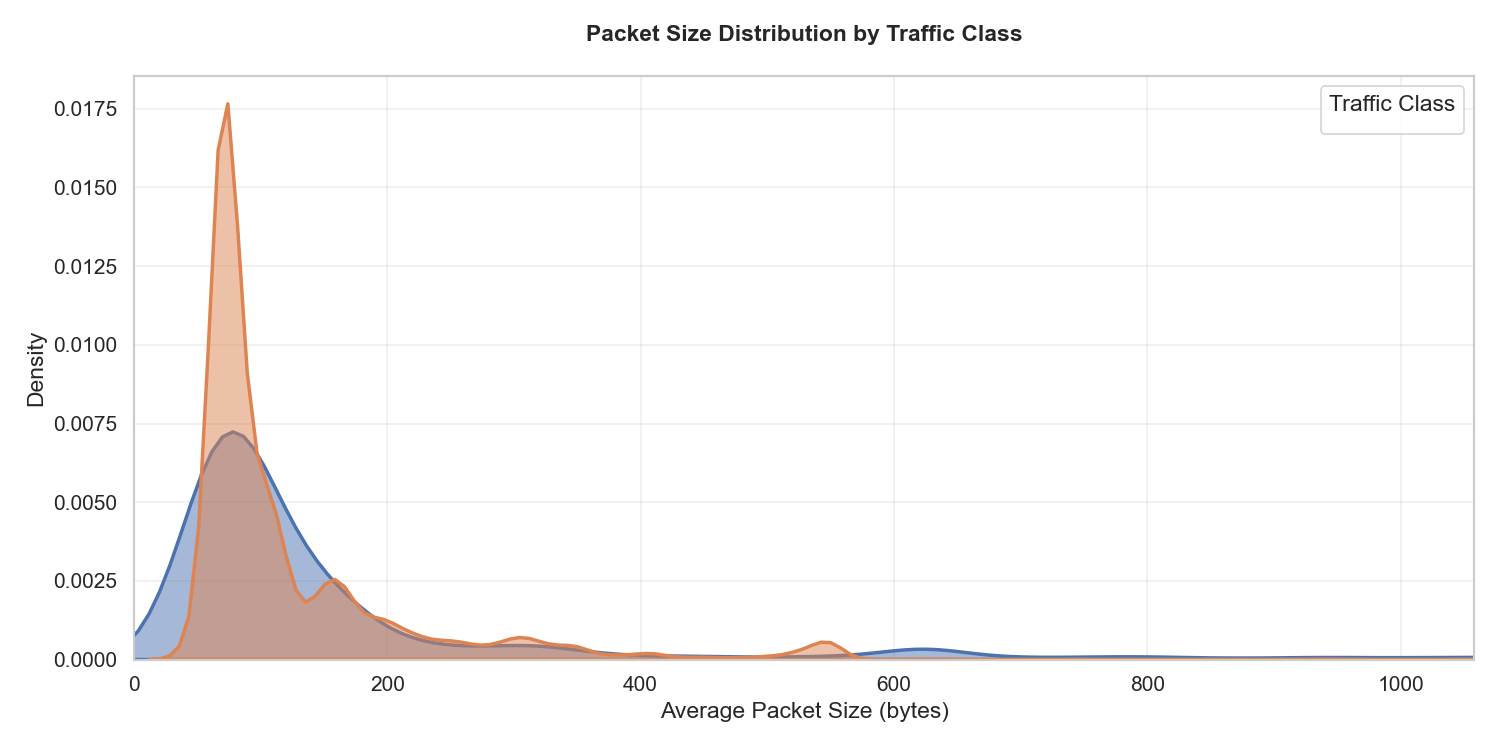
**Port Activity Heatmap**



**Description:** A heatmap showing the percentage of malicious vs normal flows for the top 15 most used destination ports.

**Insight:** Identifies which ports are more prone to botnet traffic, useful for setting network monitoring priorities.

**Packet Size Distribution:**

****

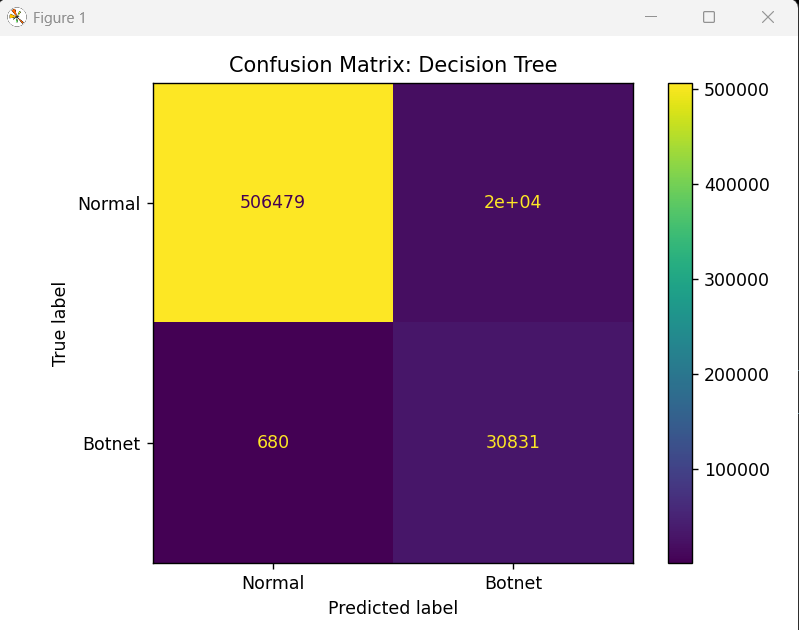
Description:  
 Kernel Density Estimation (KDE) plot for average packet sizes in each traffic class, showing how packet sizes vary and overlap.

Insight:  
 Malicious traffic favor certain packet sizes, aiding in behavior-based detection techniques.

Testing Models

**Testing Models**

**DecisionTreeClassifier**

****

**precision recall f1-score support**

**Normal 1.00 0.96 0.98 525999**

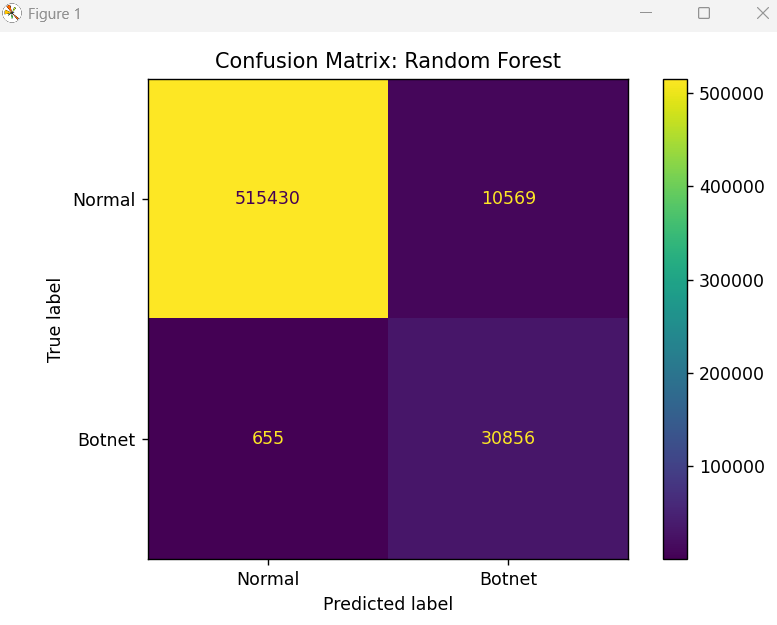
**Botnet 0.61 0.98 0.75 31511**

**accuracy 0.96 557510**

**macro avg 0.81 0.97 0.87 557510**

**weighted avg 0.98 0.96 0.97 557510**

**RandomForestClassification**

****

**Training Random Forest...**

**precision recall f1-score support**

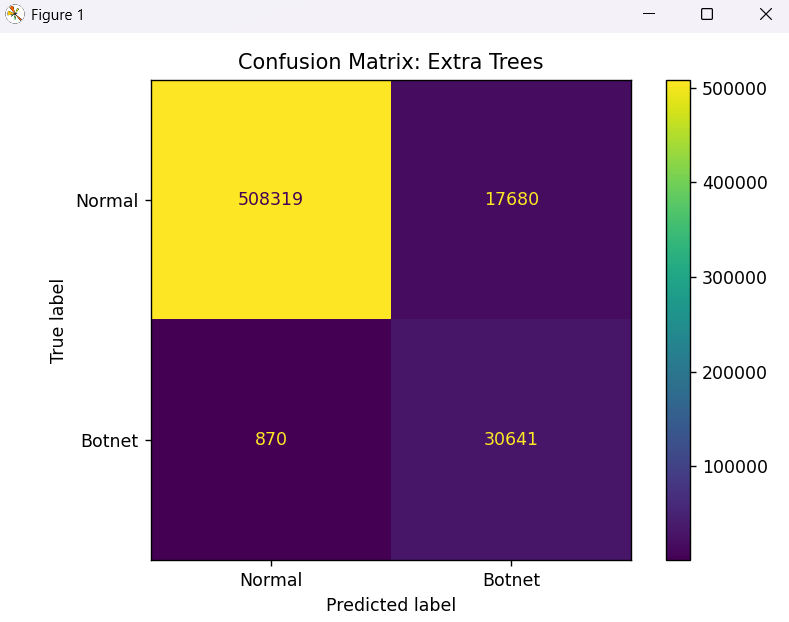
**Normal 1.00 0.98 0.99 525999**

**Botnet 0.74 0.98 0.85 31511**

**accuracy 0.98 557510**

**macro avg 0.87 0.98 0.92 557510**

**weighted avg 0.98 0.98 0.98 557510**

**ExtraTreesClassification  
**

**Training Extra Trees...**

**precision recall f1-score support**

**Normal 1.00 0.97 0.98 525999**

**Botnet 0.63 0.97 0.77 31511**

**accuracy 0.97 557510**

**macro avg 0.82 0.97 0.87 557510**

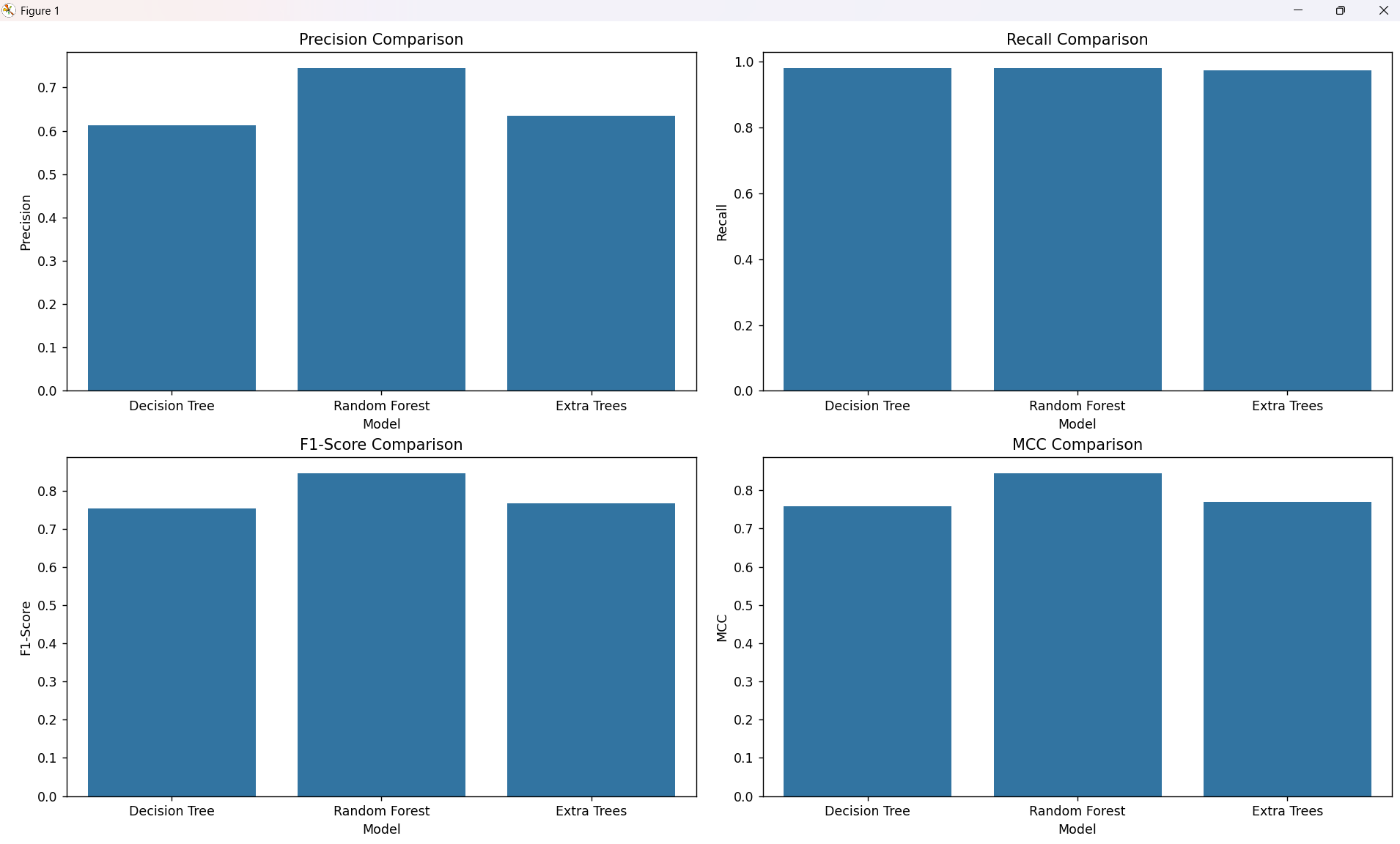
**weighted avg 0.98 0.97 0.97 557510**

Comparing Model Results

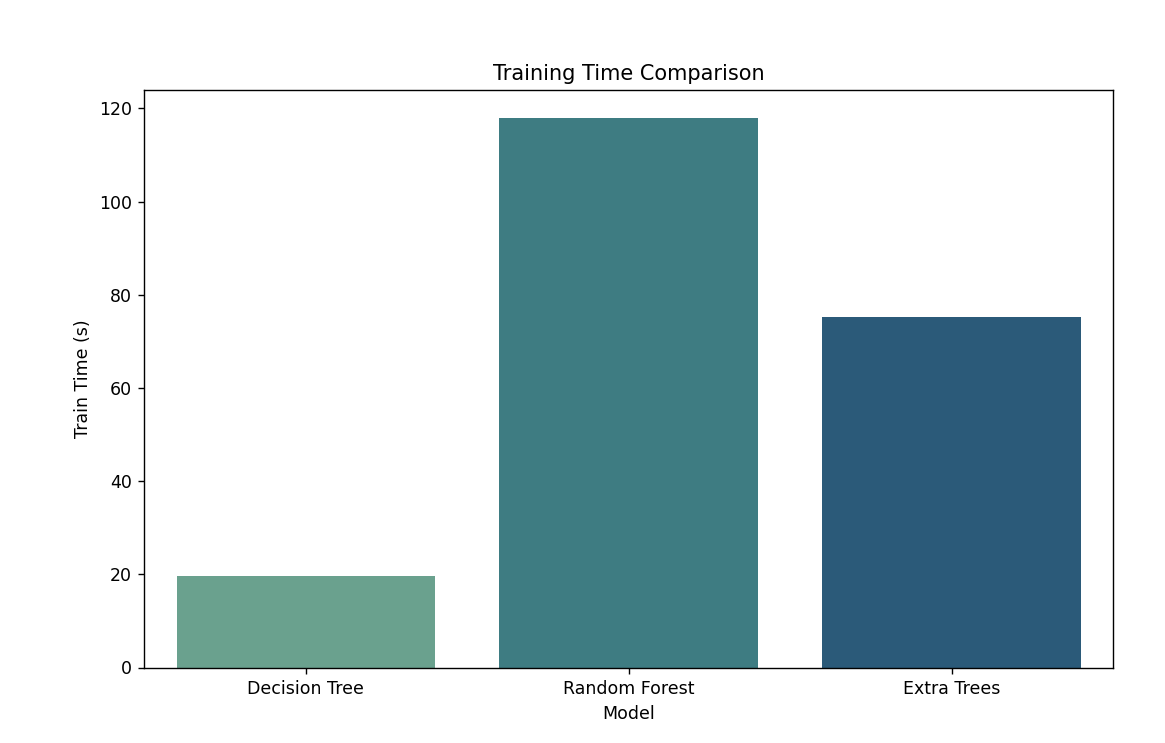
**Comparing Model Results**

Decision\_Tree v/s Random Forest v/s Extra Trees

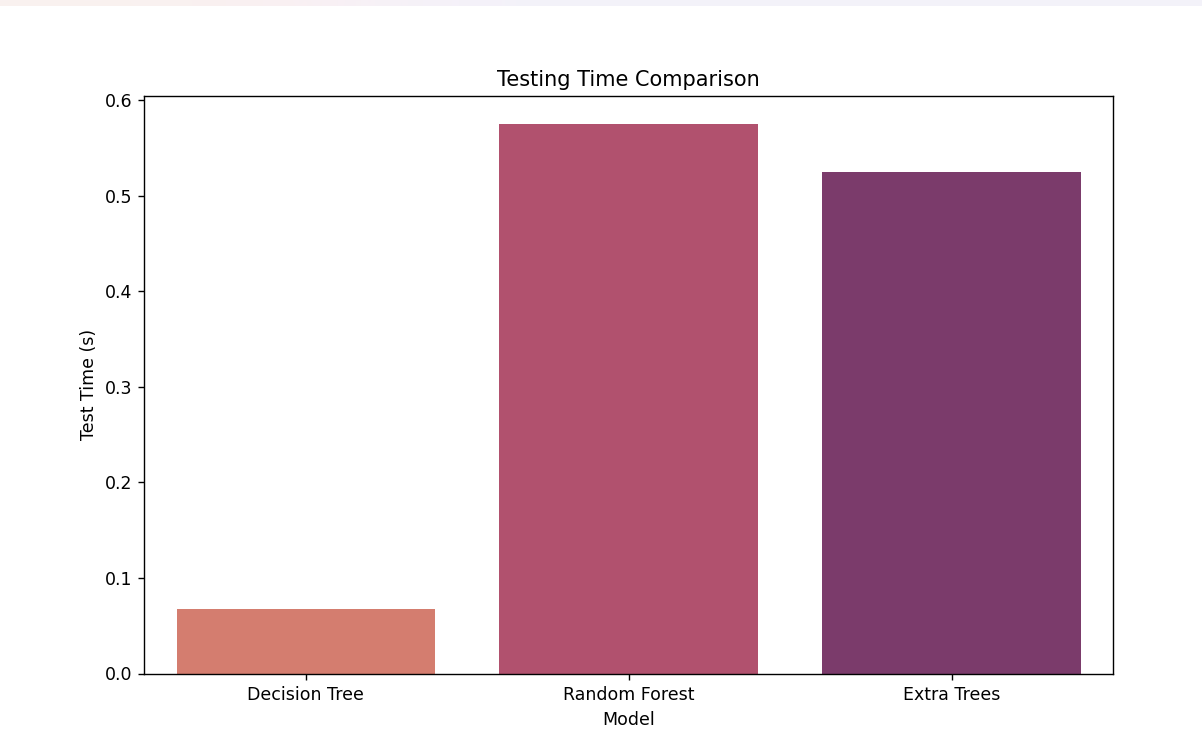
* Metrics Comparison



* Training Time



* Testing Time



Fine Tuning

**Fine Tuning**

Training Random Forest models at varying depths...

Depth 1: Train F1 = 0.7469, Test F1 = 0.2147

Depth 3: Train F1 = 0.9200, Test F1 = 0.5539

Depth 5: Train F1 = 0.9517, Test F1 = 0.6083

Depth 7: Train F1 = 0.9615, Test F1 = 0.6679

Depth 9: Train F1 = 0.9725, Test F1 = 0.7383

Depth 11: Train F1 = 0.9749, Test F1 = 0.7507

Depth 13: Train F1 = 0.9807, Test F1 = 0.7978

Depth 15: Train F1 = 0.9840, Test F1 = 0.8170

Depth 17: Train F1 = 0.9865, Test F1 = 0.8252

Depth 19: Train F1 = 0.9890, Test F1 = 0.8270

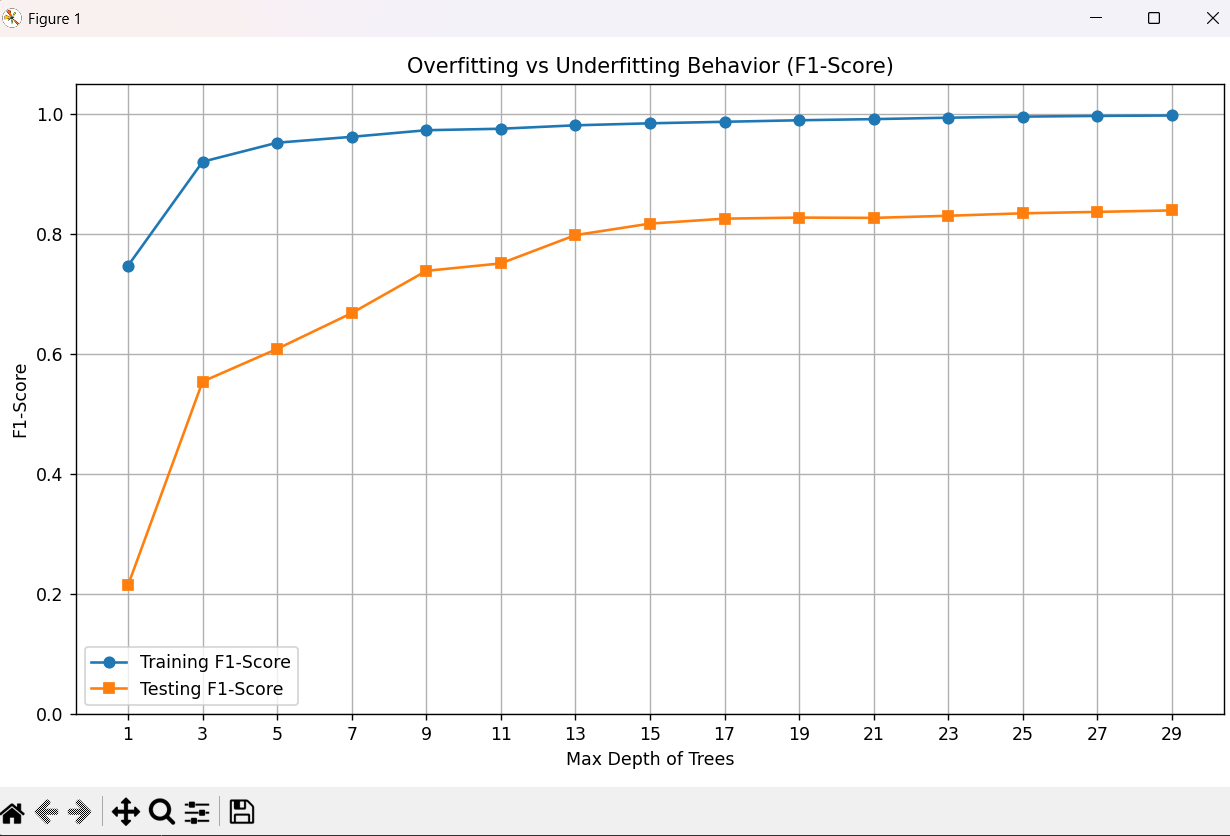
Depth 21: Train F1 = 0.9910, Test F1 = 0.8265

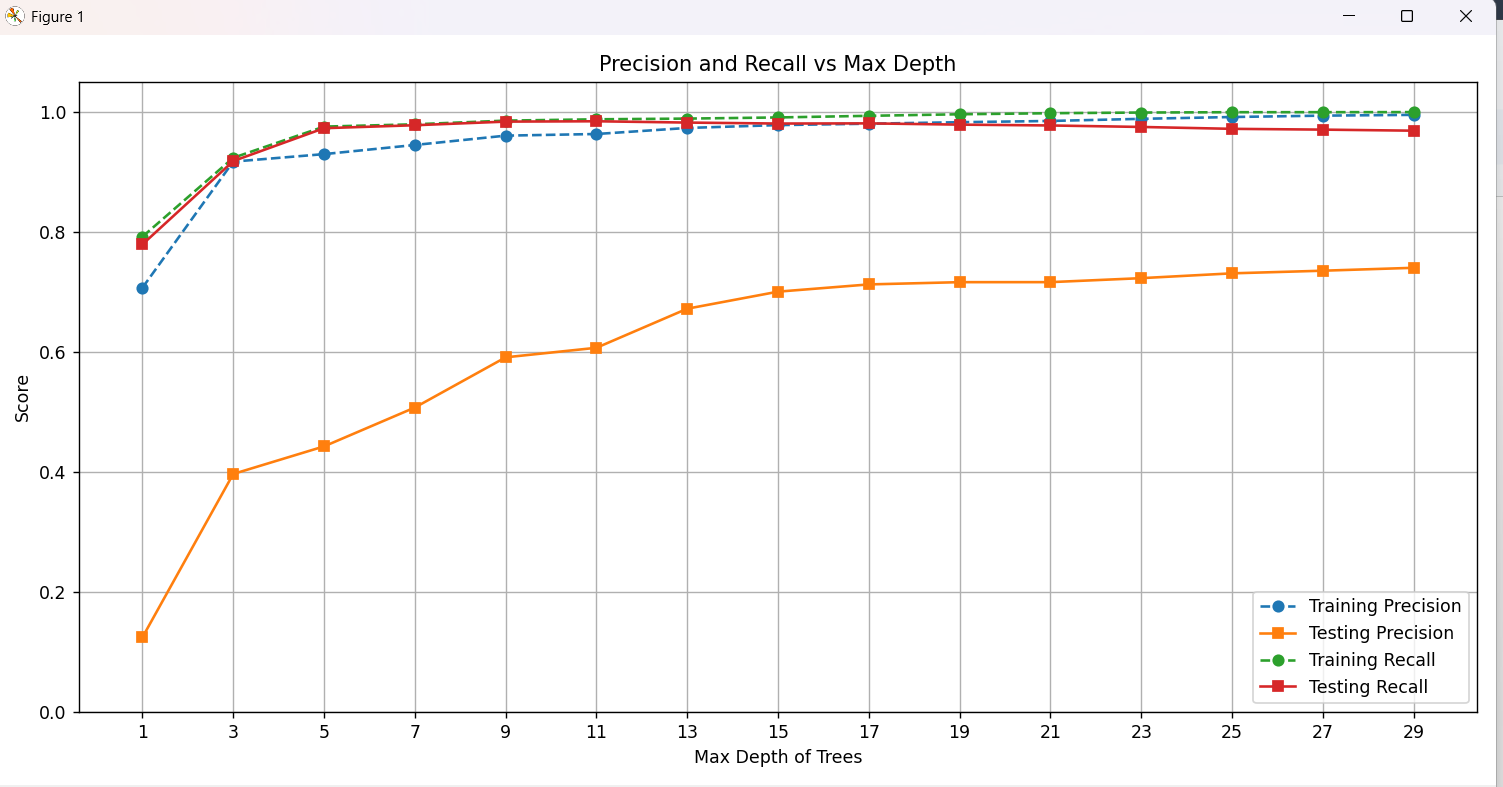
Depth 23: Train F1 = 0.9933, Test F1 = 0.8300

Depth 25: Train F1 = 0.9951, Test F1 = 0.8341

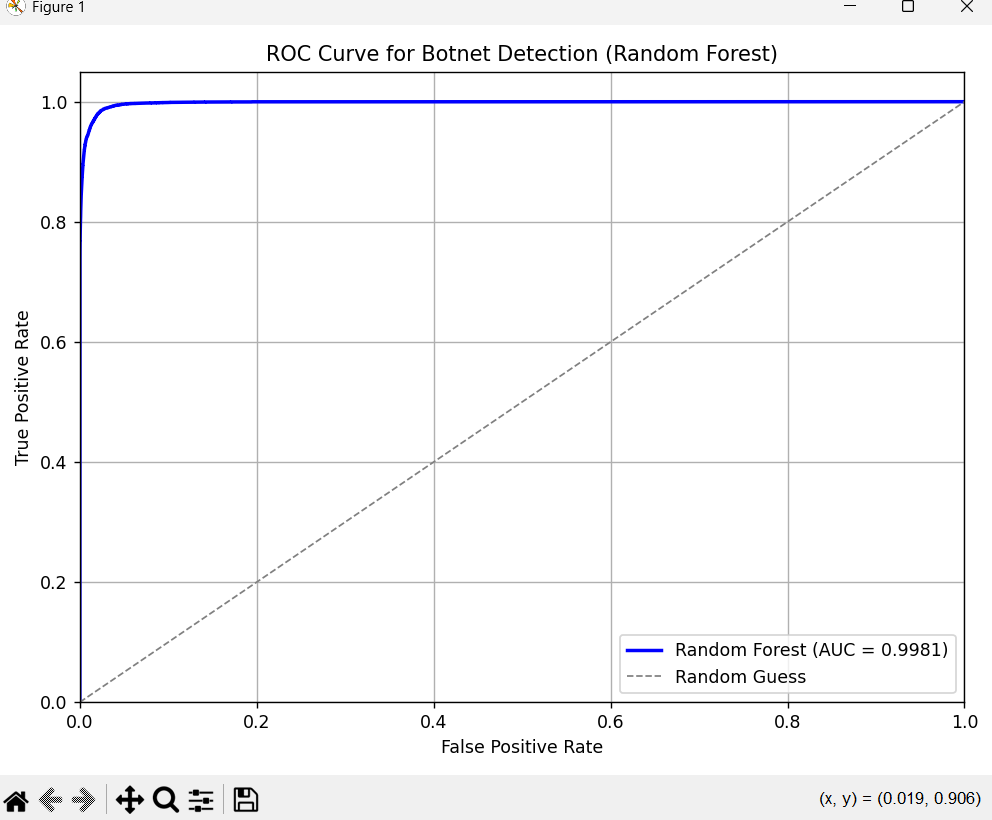
Depth 27: Train F1 = 0.9963, Test F1 = 0.8365

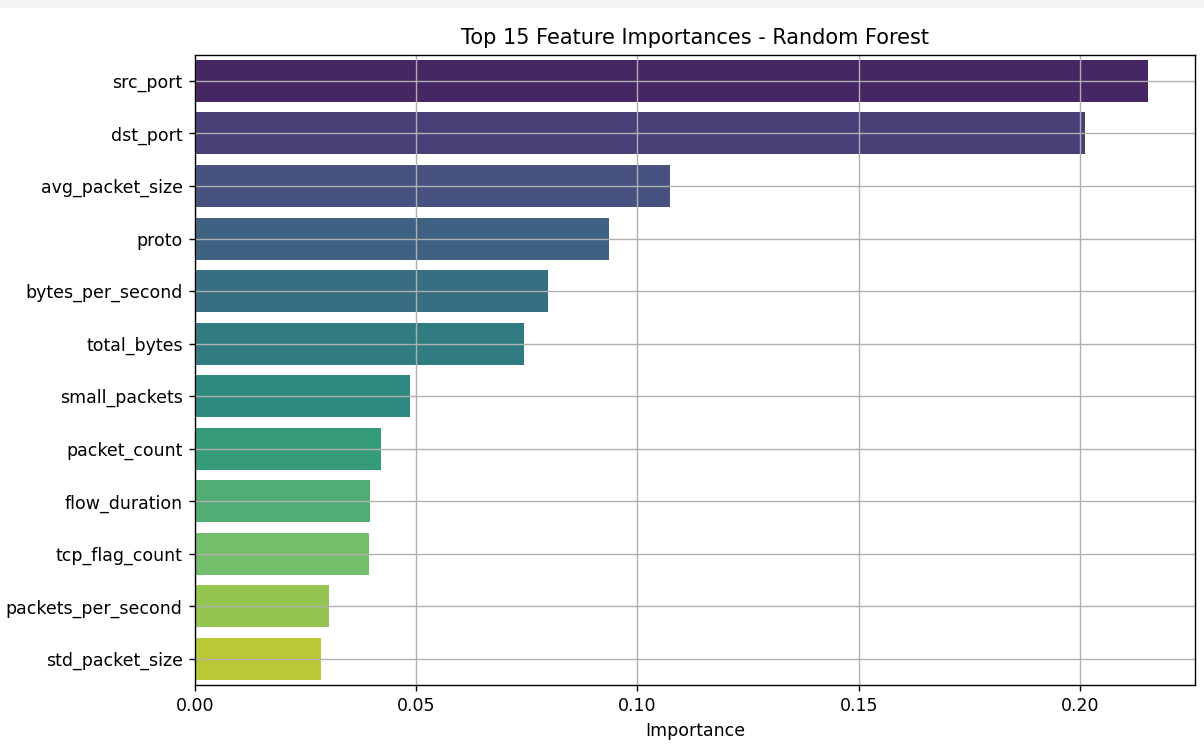
Depth 29: Train F1 = 0.9970, Test F1 = 0.8389





ROC Curve





Classification Report:

precision recall f1-score support

Normal 1.00 0.98 0.99 525999

Botnet 0.74 0.98 0.85 31511

accuracy 0.98 557510

macro avg 0.87 0.98 0.92 557510

weighted avg 0.98 0.98 0.98 557510