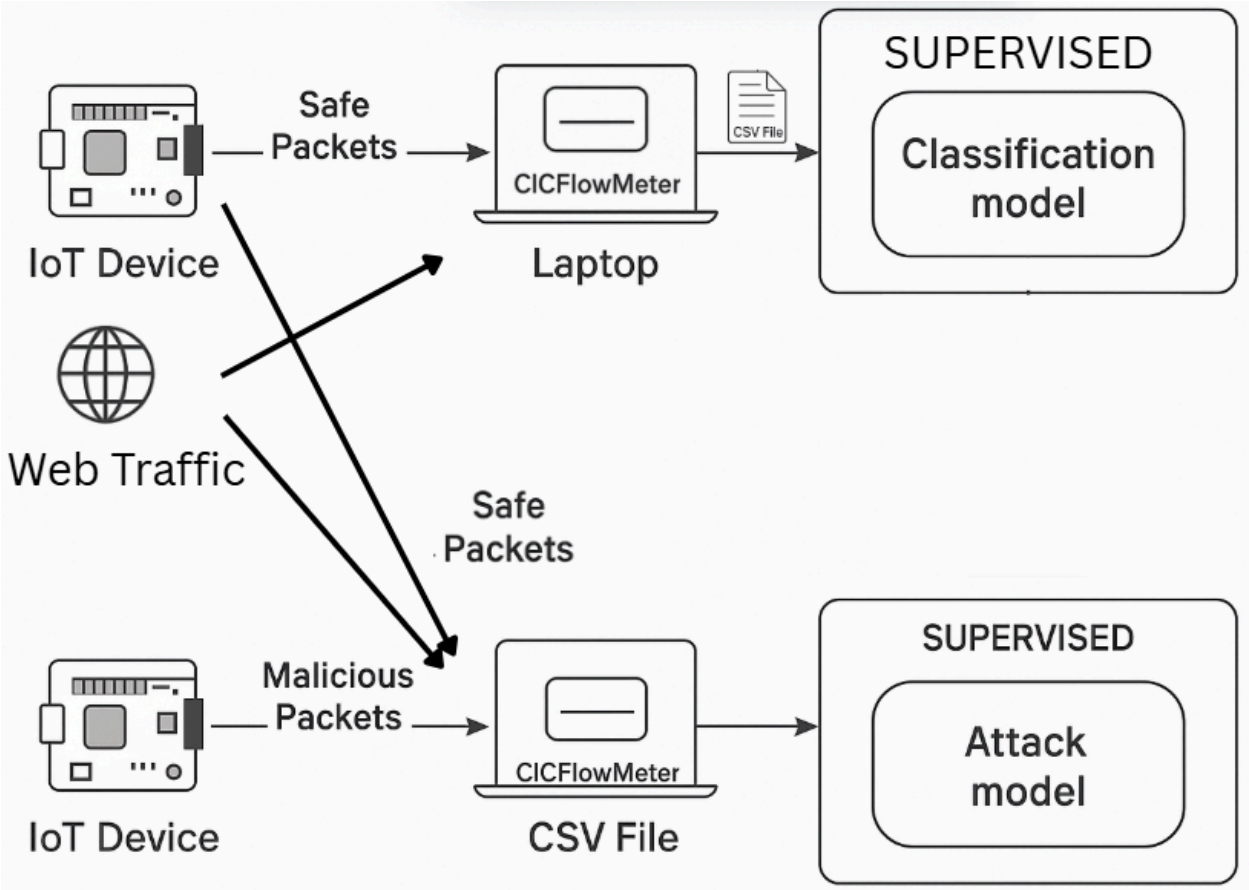
Iot Attack Guard Documentation

Ahmad Imran 21L-5374 - Zaki Qasim 21L-7642 - Abdur Rehman Bin Masud 21L-5350

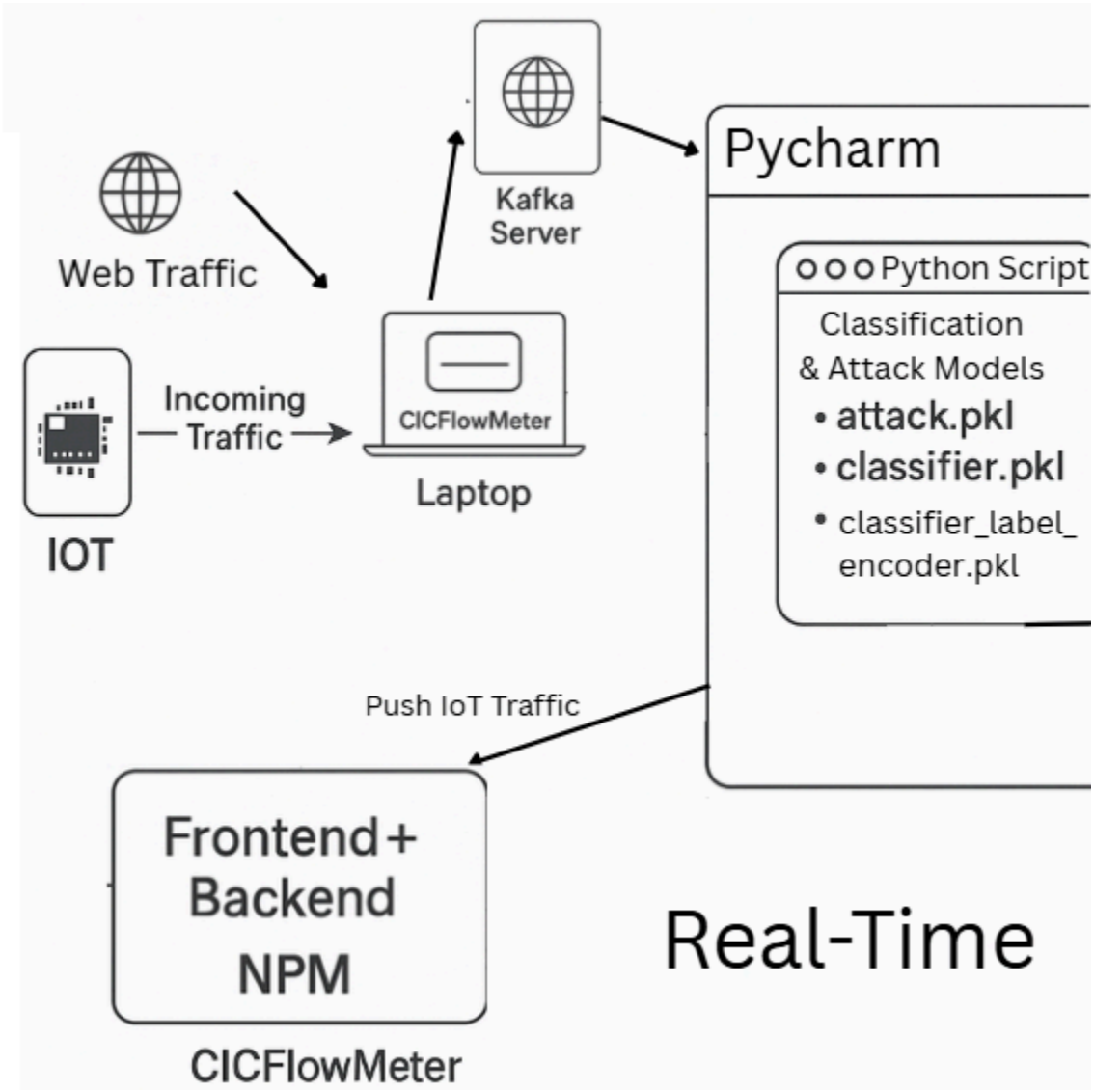
Introduction

In our IoT security system, both the IoT device classification model and the attack detection model were developed using **supervised learning techniques**. We trained these models on labeled datasets where each network flow was already categorized—for example, as either an "IoT Device" or "Non-IoT Device" in the classification task, and "Normal" or "Malicious" in the attack detection task. The training and evaluation were performed on a **Raspberry Pi Model 3B**, showcasing the lightweight and real-time capability of our models. Among several tested algorithms, we selected **XGBoost** due to its high accuracy, fast computation, and robustness against overfitting. During experimentation, XGBoost consistently outperformed other classifiers like Logistic Regression and Decision Trees in terms of **precision, recall, F1-score**, and overall classification accuracy. This made it an ideal choice for both detecting abnormal behavior in IoT traffic and classifying the type of device on the network.

Training and Testing Setup



Integration Setup



CICFlowMeter Setup & Packet Capture Documentation

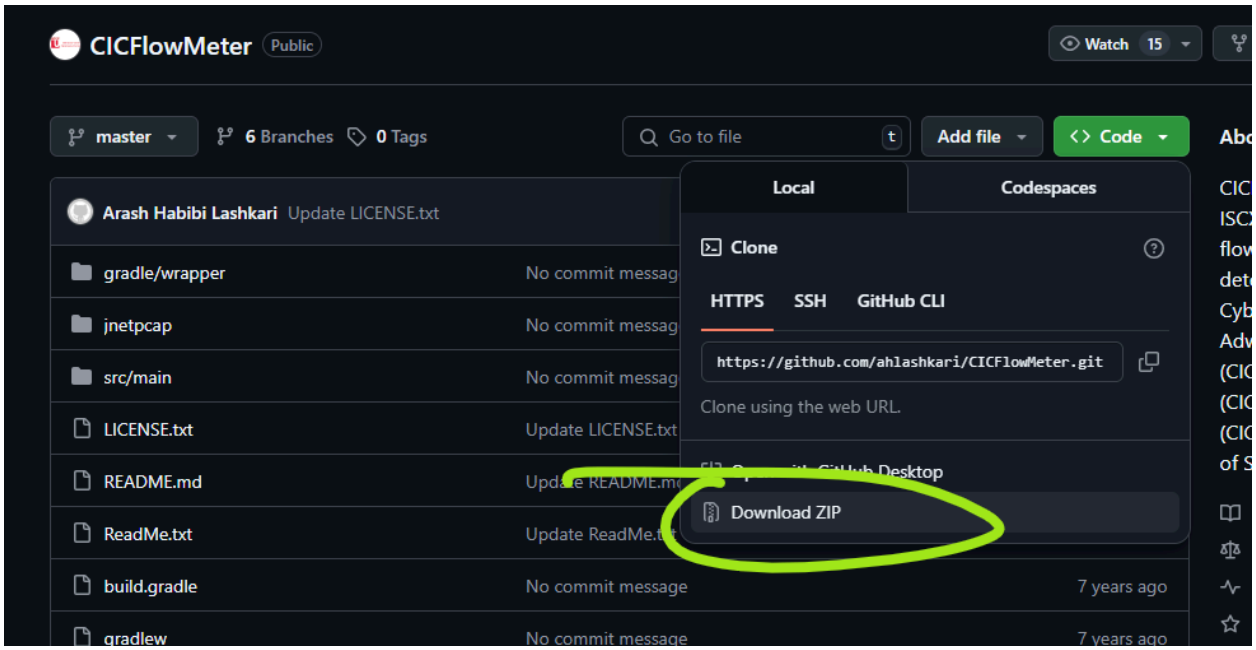
Download Eclipse

Download and install Eclipse IDE for windows from https://www.eclipse.org/downloads/

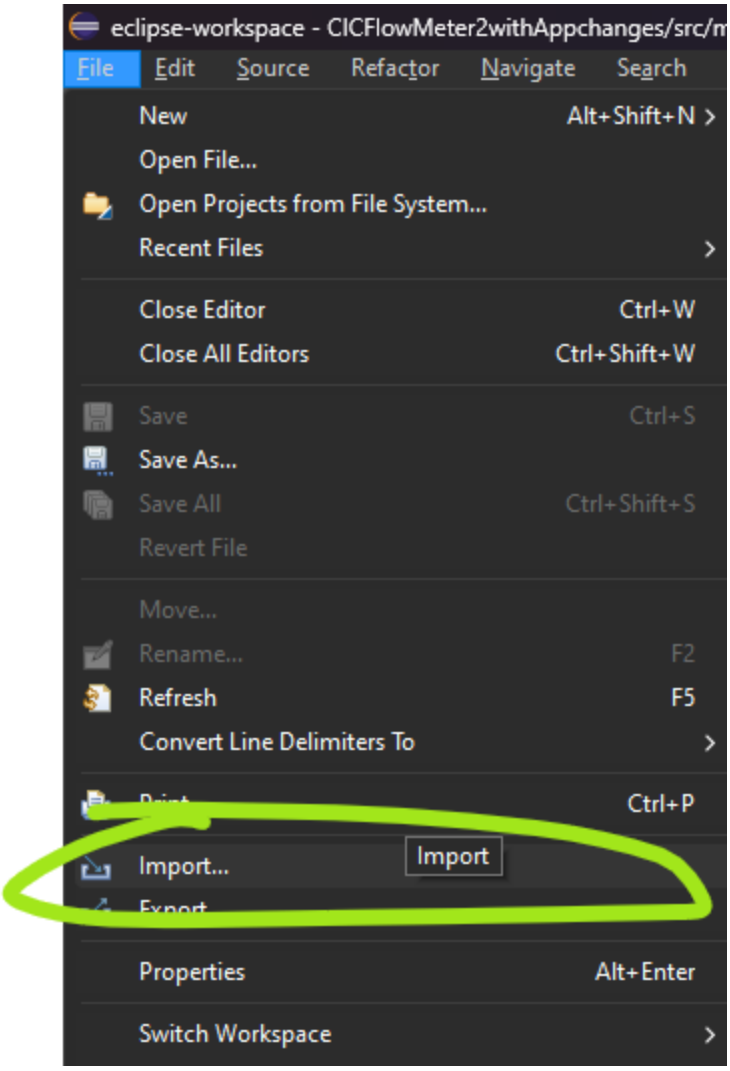
For Linux:
https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-12/R/eclipse-java-2024-12-R-linux-gtk-x86_64.tar.gz

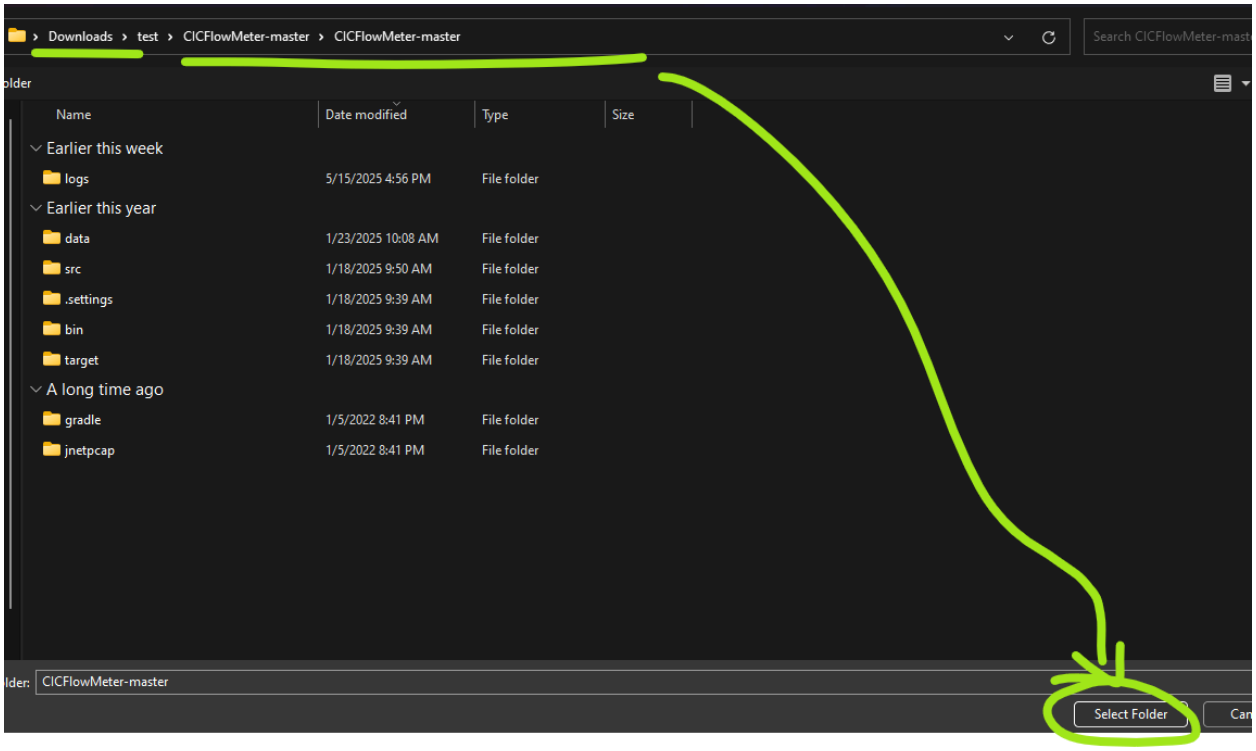
Downloading and Importing CICFlowMeter

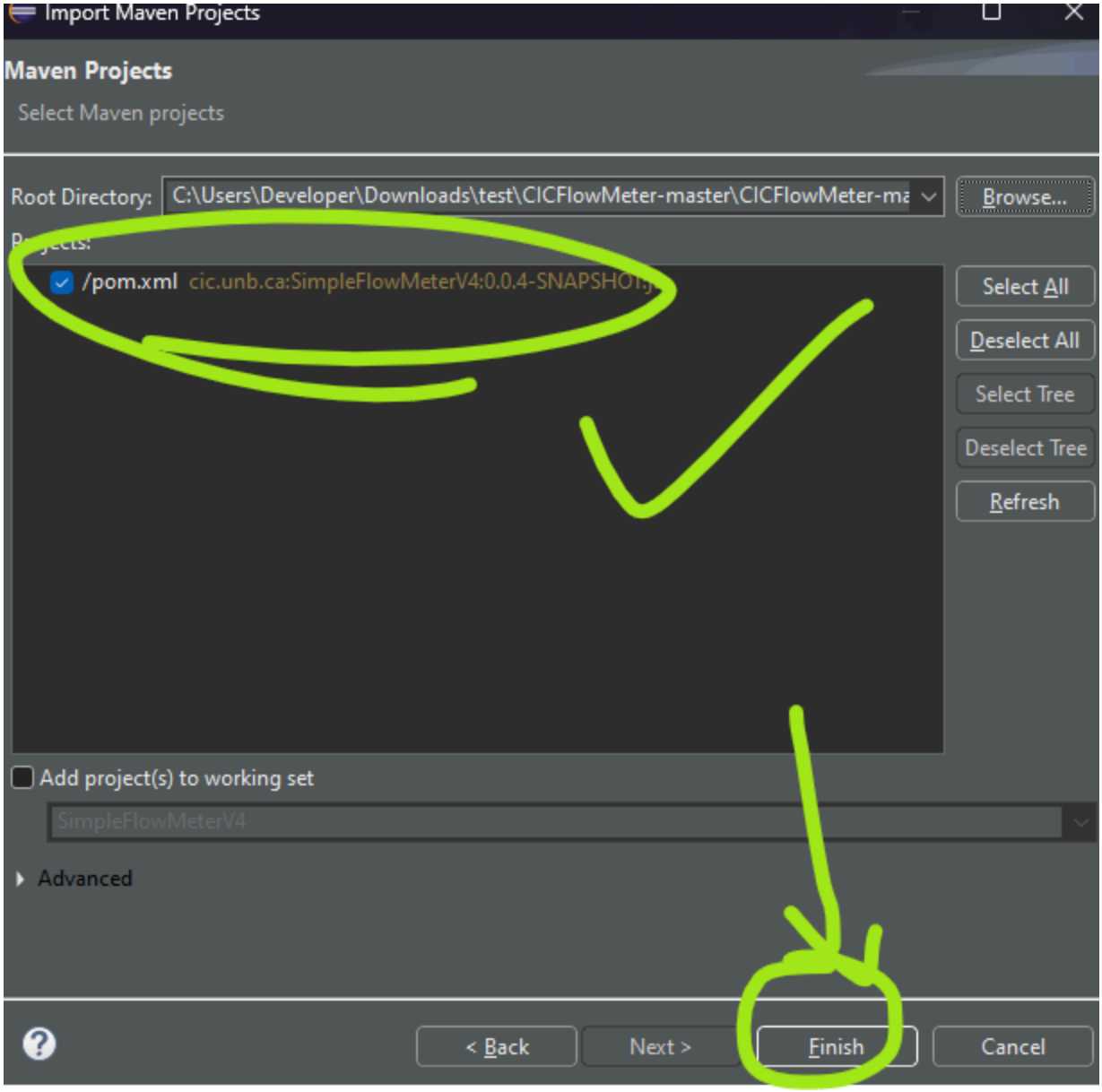
- We downloaded the official CICFlowMeter from Github: https://github.com/ahlashkari/CICFlowMeter



- Download and extract the zip file and import it into **Eclipse IDE** as a Java project.

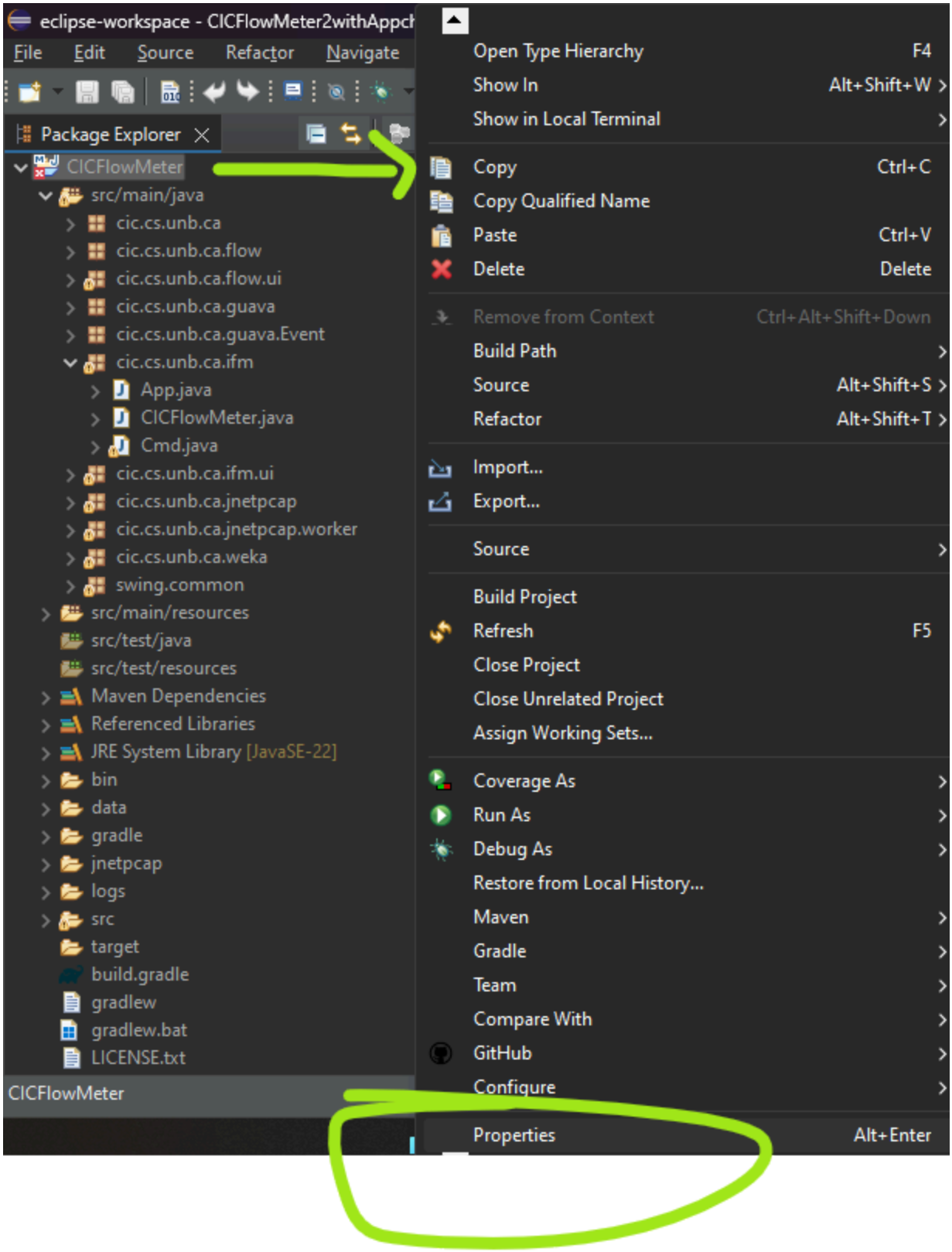

eclipse-workspace - CICFlowMeter2withAppchanges/src/m
File
Edit
Source
Refactor
Navigate
Search
New
Alt+Shift+N
Open File...
Open Projects from File System...
Recent Files
Close Editor
Ctrl+W
Close All Editors
Ctrl+Shift+W
Save
Ctrl+S
Save As...
Save All
Ctrl+Shift+S
Revert File
Move...
Rename...
F2
Refresh
F5
Convert Line Delimiters To
Ctrl+P
Import...
Import
Properties
Alt+Enter
Switch Workspace

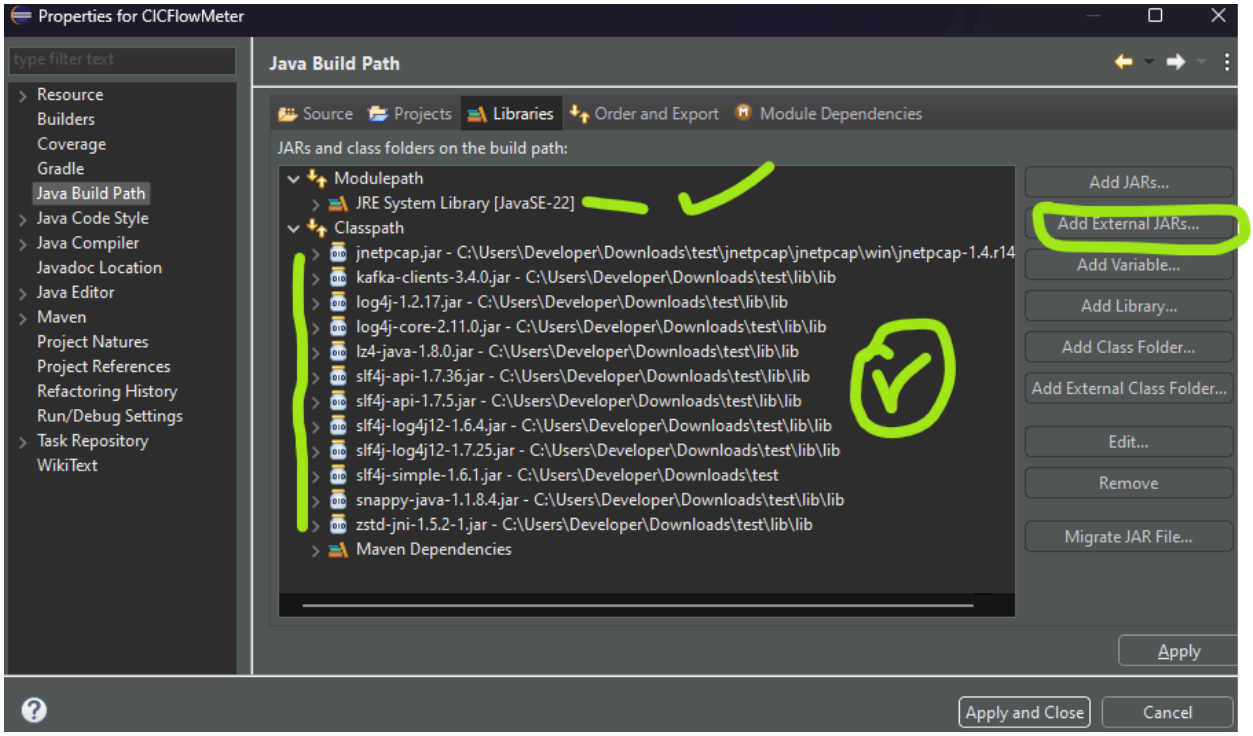

Downloads
test
CICFlowMeter-master
CICFlowMeter-master
Search CICFlowMeter-mast
Name
Date modified
Type
Size
Earlier this week
logs
5/15/2025 4:56 PM
File folder
Earlier this year
data
1/23/2025 10:08 AM
File folder
src
1/18/2025 9:50 AM
File folder
.settings
1/18/2025 9:39 AM
File folder
bin
1/18/2025 9:39 AM
File folder
target
1/18/2025 9:39 AM
File folder
A long time ago
gradle
1/5/2022 8:41 PM
File folder
jnetpcap
1/5/2022 8:41 PM
File folder
CICFlowMeter-master
Select Folder



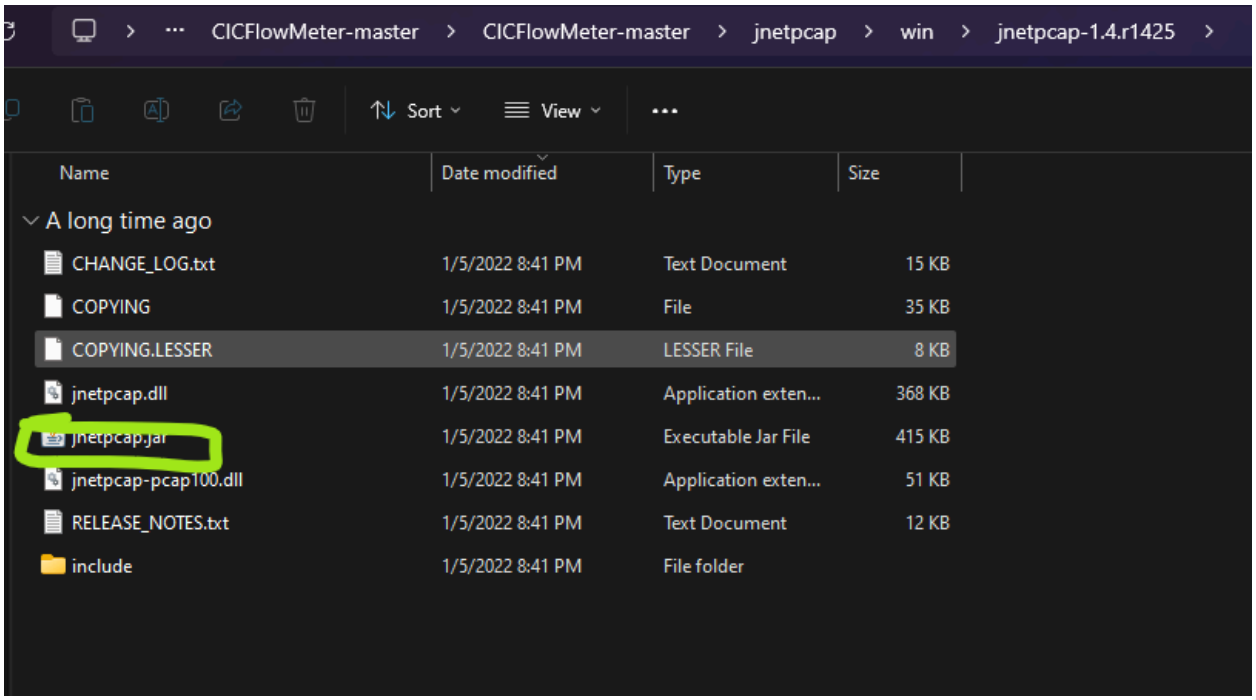
Step 2: Adding Required Dependencies

- After importing the project, we added all required Java dependencies manually in Eclipse.


eclipse-workspace - CICFlowMeter2withAppch
File
Edit
Source
Refactor
Navigate
Package Explorer
CICFlowMeter
src/main/java
cic.cs.unb.ca
cic.cs.unb.ca.flow
cic.cs.unb.ca.flow.ui
cic.cs.unb.ca.guava
cic.cs.unb.ca.guava.Event
cic.cs.unb.ca.ifm
App.java
CICFlowMeter.java
Cmd.java
cic.cs.unb.ca.ifm.ui
cic.cs.unb.ca.jnetpcap
cic.cs.unb.ca.jnetpcap.worker
cic.cs.unb.ca.weka
swing.common
src/main/resources
src/test/java
src/test/resources
Maven Dependencies
Referenced Libraries
JRE System Library [JavaSE-22]
bin
data
gradle
jnetpcap
logs
src
target
build.gradle
gradlew
gradlew.bat
LICENSE.txt
CICFlowMeter
Open Type Hierarchy F4
Show In Alt+Shift+W
Show in Local Terminal
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Source
Build Project
Refresh F5
Close Project
Close Unrelated Project
Assign Working Sets...
Coverage As
Run As
Debug As
Restore from Local History...
Maven
Gradle
Team
Compare With
GitHub
Configure
Properties Alt+Enter



- Any missing libraries (e.g., `jNetPcap`) were added to the build path to ensure the project compiled successfully.
- Use jnetpcap.jar obtained from the win folder, otherwise for linux select the linux folder



- All other jar files can be downloaded from https://mvnrepository.com/ and Apache log4j 1.2 - Download Apache log4j 1.2 and https://www.slf4j.org/download

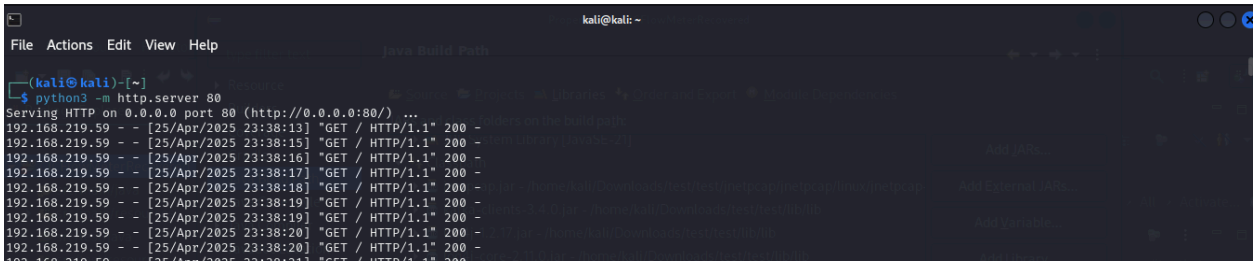
Step 3: Running CICFlowMeter

- We ran the CICFlowMeter project directly within Eclipse.
- In some cases, CICFlowMeter did **not capture any traffic.**.

⚠️ Important Note on Packet Capture Behavior

- To ensure successful packet capture:
    - **Reboot the system (laptop).**
    - **After reboot**, avoid opening any browsers or initiating network activity.
    - Go **straight to the Kali Linux VM** and run CICFlowMeter.
    - This method helps ensure that CICFlowMeter will capture packets..

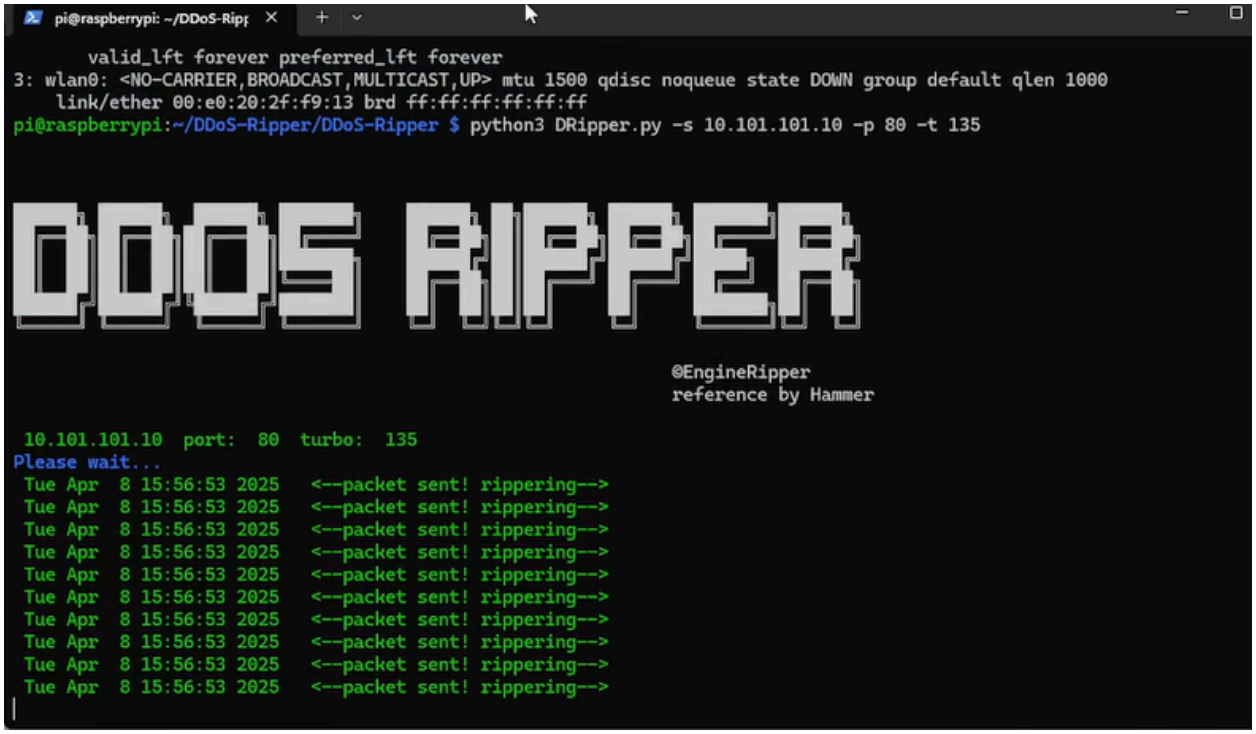
We opened Python http server on port 80 to perform the ddos attack on.

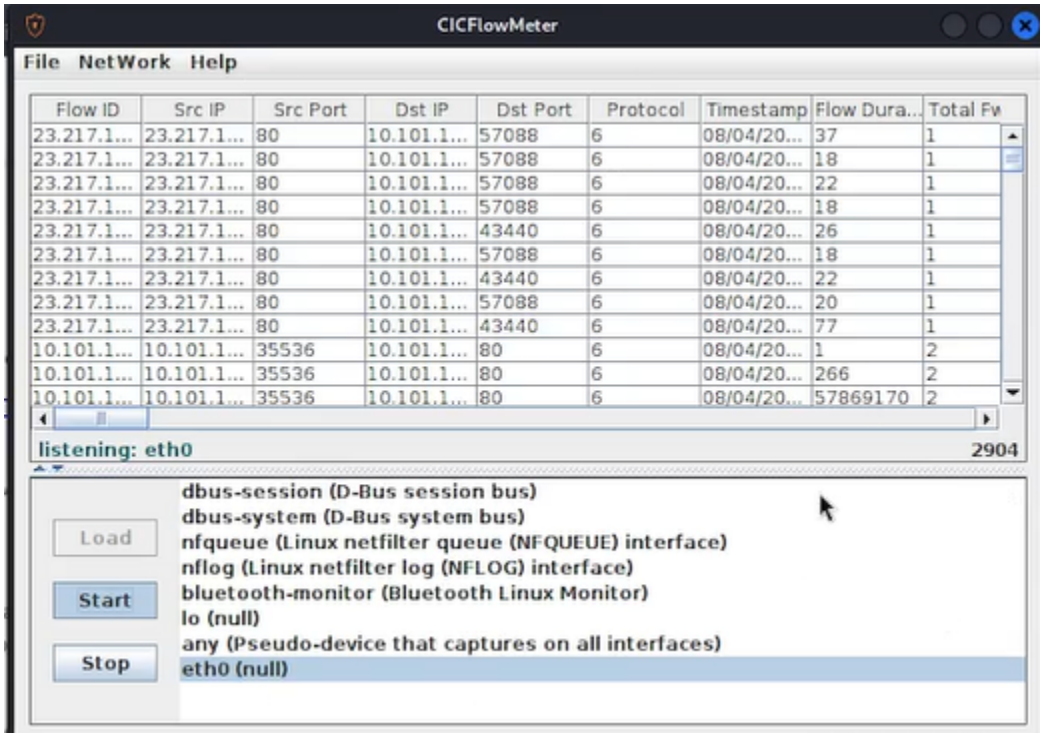


Step 4: Generating Malicious and Benign Traffic

To simulate **DDoS traffic**, we connected to a Raspberry Pi via **SSH** and ran `ddos-ripper`:

```
 ssh pi@<raspberry_ip>
cd ddos-ripper
python3 ddos-ripper.py -s <target_ip> -p <port>
```



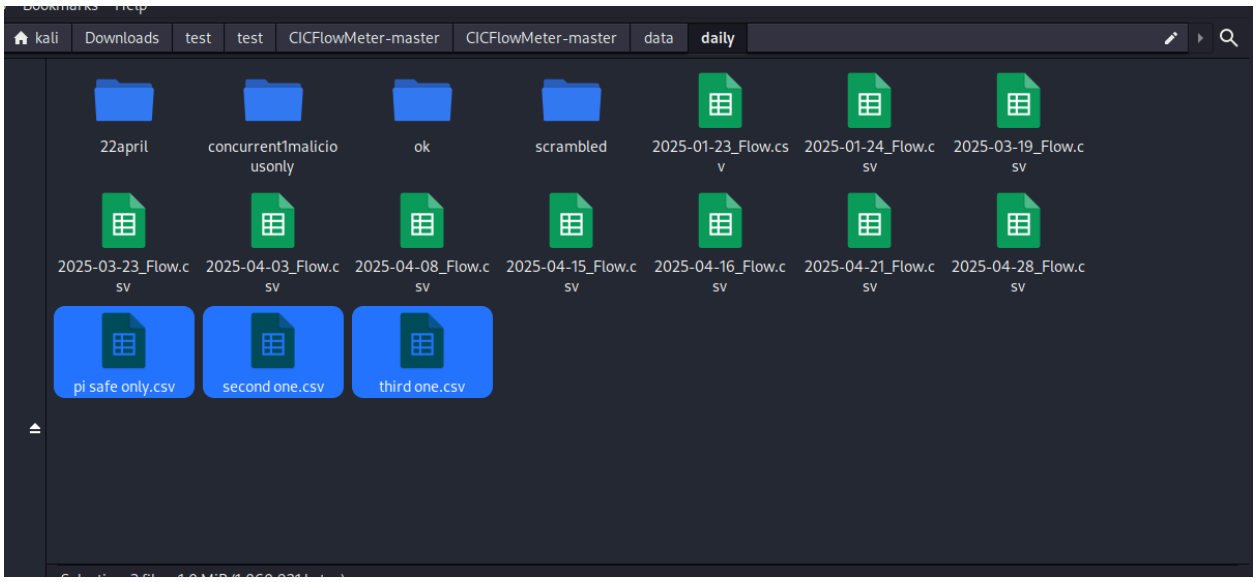


- During the packet capture session, we also generated **normal traffic** such as:
    - Downloading a file from **Google Drive**.

- Browsing safe websites.
- Generating safe traffic from the IoT-Device like Raspberry PI.

Step 5: Capturing Flow Data

- CICFlowMeter converted the captured packets into flow-based CSV files.



- These files were used later for labeling and model training.

IoT Traffic Classification Pipeline: Data Preparation, Model Training, and Inference

Excel to CSV Conversion

The first step involves converting an Excel spreadsheet into a CSV format, which is more suitable for most data processing pipelines. The file `modeldata.xlsx` is read using the `pandas` library, and its contents are saved to a new file with a `.csv` extension. The original filename is preserved, with only the extension replaced. This transformation facilitates downstream compatibility with tools that do not support Excel files natively.

```python
import pandas as pd

# Load Excel file
xlsx_path = 'modeldata.xlsx'  # Change to your actual filename
df = pd.read_excel(xlsx_path)

# Save as CSV
csv_path = xlsx_path.replace('.xlsx', '.csv')
df.to_csv(csv_path, index=False)

print(f"Converted and saved to: {csv_path}")
```

Label Assignment Based on Source IP

This section prepares the dataset for supervised learning by assigning a `Label` to each record based on the source IP address. The IPs `192.168.219.59` and `192.168.195.59` are explicitly identified as IoT devices, while all others are treated as non-IoT. The labeling is performed using a lambda function applied to the `Src IP` column. Once labeled, the updated dataset is saved back to the original file, ensuring the changes are reflected immediately.

```python
import pandas as pd
# Load the CSV file
df = pd.read_csv('classdata.csv')
# Add or update the 'Label' column based on 'Src IP'
df['Label'] = df['Src IP'].apply(lambda ip: 'IoT Device' if ip == '192.168.219.59' or ip ==
'192.168.195.59' else 'Non-IoT Device')
# Save changes to the same file (overwrite)
df.to_csv('classdata.csv', index=False)
print("Label column updated in 'bothdata.csv'")
```

Training and Inference Pipeline Using XGBoost

This part establishes a complete training and prediction pipeline for classifying network packets. The dataset is first cleaned by removing fields such as IP addresses and port numbers, which are not relevant for classification. Categorical features are one-hot encoded, and missing or infinite values are replaced to ensure model compatibility.

The target labels are encoded using `LabelEncoder`, and the dataset is split into training and validation sets. An `XGBClassifier` is then trained with standard parameters.

Once trained, the model is evaluated on the validation set using classification metrics. The pipeline proceeds to inference on a separate dataset (`malicious_only.csv`). The test data is preprocessed to match the structure of the training data, and predictions are generated along with confidence scores.

To enhance reliability, only rows with a confidence score of at least 0.967 are retained. The filtered results include both predicted labels and their associated confidence, which are saved to disk and summarized for reporting.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from xgboost import XGBClassifier
from sklearn.metrics import classification_report

# === UTILITIES ===
def clean_df(df):
    # Replace infinite values with NaN, then fill NaNs with 0 (or you could df.dropna())
    return df.replace([np.inf, -np.inf], np.nan).fillna(0)

# === TRAINING ===
train_df = pd.read_csv('classdata.csv')

# Drop IP and Port columns
drop_cols = ['Src IP', 'Dst IP', 'Src Port', 'Dst Port']
train_df = train_df.drop(columns=[c for c in drop_cols if c in train_df.columns])

# Features and labels
X = train_df.drop('Label', axis=1)
y = train_df['Label']

# One-hot encoding
X_encoded = pd.get_dummies(X)

# Clean any infinities / NaNs
X_encoded = clean_df(X_encoded)

# Label encoding
le = LabelEncoder()
y_encoded = le.fit_transform(y)

# Train-test split
X_train, X_val, y_train, y_val = train_test_split(
    X_encoded, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded
)

# XGBoost classifier
model = XGBClassifier(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    use_label_encoder=False,
    eval_metric='mlogloss',
    random_state=42
)
model.fit(X_train, y_train)
```

```
# Evaluate
val_preds = model.predict(X_val)
print("Validation Report:\n",
      classification_report(y_val, val_preds, target_names=le.classes_))

# === PREDICTION ===
test_path = 'malicious_only.csv'
test_df = pd.read_csv(test_path)

# Clean and align test features
test_df_clean = test_df.drop(columns=[c for c in drop_cols if c in test_df.columns],
errors='ignore')
test_encoded = pd.get_dummies(test_df_clean)

# Ensure same columns as training
test_encoded = test_encoded.reindex(columns=X_encoded.columns, fill_value=0)

# Clean infinities / NaNs in test set
test_encoded = clean_df(test_encoded)

# Predict classes and probabilities
pred_probs = model.predict_proba(test_encoded)
pred_classes = model.predict(test_encoded)

# Decode labels and attach confidence
test_df['Label'] = le.inverse_transform(pred_classes)
test_df['Confidence'] = pred_probs.max(axis=1)

# Filter: keep only high-confidence rows
filtered_df = test_df[test_df['Confidence'] >= 0.967].copy()

# Count IoT vs Non-IoT
counts = filtered_df['Label'].value_counts()
iot_count = counts.get('IoT', 0)
non_iot_count = counts.sum() - iot_count

print(f"Total high-confidence packets: {len(filtered_df)}")
print(f"IoT packets: {iot_count}")
print(f"Non-IoT packets: {non_iot_count}")

# Save filtered predictions
filtered_df.to_csv(test_path, index=False)
print(f"Filtered predictions (confidence ≥ 0.967) saved to: {test_path}")
filtered_df['Label'].value_counts()
```

Training the Model and Saving Artifacts

This segment focuses on training the XGBoost model and saving its artifacts for later use. The training data is loaded and preprocessed in the same way as described earlier—irrelevant columns are removed, features are encoded, and missing values are handled.

After training, the model is serialized using `joblib`, along with the label encoder. These saved objects (`classifier.pkl` and `classifier_label_encoder.pkl`) can be used in a separate environment to perform inference, eliminating the need for retraining.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from xgboost import XGBClassifier
from joblib import dump

# Utility to clean infinities/NaNs
def clean_df(df):
    return df.replace([np.inf, -np.inf], np.nan).fillna(0)

# 1. Load and preprocess training data
train_df = pd.read_csv('classdata.csv')
drop_cols = ['Src IP', 'Dst IP', 'Src Port', 'Dst Port']
train_df = train_df.drop(columns=[c for c in drop_cols if c in train_df.columns])

X = train_df.drop('Label', axis=1)
y = train_df['Label']

X_encoded = pd.get_dummies(X)
X_encoded = clean_df(X_encoded)

le = LabelEncoder()
y_encoded = le.fit_transform(y)

X_train, X_val, y_train, y_val = train_test_split(
    X_encoded, y_encoded,
    test_size=0.2, random_state=42, stratify=y_encoded
)

# 2. Train the model
model = XGBClassifier(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    use_label_encoder=False,
    eval_metric='mlogloss',
    random_state=42
)
model.fit(X_train, y_train)

# 3. Save model and encoder
dump(model, 'classifier.pkl')
```

```
dump(le,    'classifier_label_encoder.pkl')
print("Trained model saved to classifier.pkl")
print("Label encoder saved to classifier_label_encoder.pkl")
```

Predicting on New Data Using the Saved Model

The final block loads the trained model and encoder to perform inference on a new dataset (`noiotdata.csv`). The test data undergoes the same preprocessing steps to ensure feature alignment with the training set. This includes one-hot encoding and reindexing based on the model's expected input structure.

Predictions are generated for each row, along with a confidence score. The output includes both the predicted class label and the associated confidence, and is saved back into the same CSV file. The label distribution is also printed to provide a quick overview of the results.

```
import numpy as np
import pandas as pd
from joblib import load

# Utility to clean infinities/NaNs
def clean_df(df):
    return df.replace([np.inf, -np.inf], np.nan).fillna(0)

# 1. Load model and encoder
model = load('classifier.pkl')
le    = load('classifier_label_encoder.pkl')

# 2. Load and preprocess test data
test_df = pd.read_csv('noiotdata.csv')
drop_cols = ['Src IP', 'Dst IP', 'Src Port', 'Dst Port']
test_df_clean = test_df.drop(columns=[c for c in drop_cols if c in test_df.columns],
errors='ignore')

# One-hot encode and align to training columns
X_train_dummy_cols = load('xgb_model.pkl').get_booster().feature_names  # or re-use a saved
list of train columns
test_encoded = pd.get_dummies(test_df_clean)
test_encoded = test_encoded.reindex(columns=model.get_booster().feature_names, fill_value=0)
# ensure same features

test_encoded = clean_df(test_encoded)

# 3. Predict
probs = model.predict_proba(test_encoded)
classes = model.predict(test_encoded)

test_df['Label']      = le.inverse_transform(classes)
```

```
test_df['Confidence'] = probs.max(axis=1)

# 4. Save predictions
output_path = 'noiotdata.csv'
test_df.to_csv(output_path, index=False)
print(f"Predictions saved to {output_path}")
test_df['Label'].value_counts()
```

Attack Detection Pipeline: Labeling, Visualization, Training, and Inference

Labeling Packets as Malicious or Normal

This step establishes ground truth labels based on domain knowledge. A packet is marked as `malicious` if its source IP is `192.168.195.59`, its destination IP is `192.168.195.136`, and it targets port 80 or 443. All other traffic is labeled as `normal`. Once labeled, the dataset is saved for downstream tasks, and label counts are printed for validation.

```
import pandas as pd
import numpy as np

# 1. Load your data
df = pd.read_csv('data2.csv')

# 2. Define the “malicious” condition:
#    Src IP == 192.168.168.59
# AND Dst IP == 192.168.168.136
# AND Dst Port is either 80 or 443
cond = (
    (df['Src IP']  == '192.168.195.59') &
    (df['Dst IP']  == '192.168.195.136') &
    (df['Dst Port'].isin([80, 443]))
)

# 3. Create the Label column
df['Label'] = np.where(cond, 'malicious', 'normal')

# 4. (Optional) Quick sanity check:
print(df['Label'].value_counts())

# 5. Save your newly labeled data
df.to_csv('labeled_data.csv', index=False)
```

Merging and Visualizing Network Behavior

Separate malicious and safe datasets are merged into a single time-sorted DataFrame. A `Label` column is assigned to each entry. The data is then visualized over time for various traffic features. Each plot compares how a specific feature behaves in malicious versus normal traffic, using timestamped line graphs with explicit axis labels and units. This aids in identifying temporal patterns or anomalies related to attacks.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load datasets
malicious_df = pd.read_csv('malicious_only.csv')
safe_df = pd.read_csv('safe_only.csv')

# Label datasets
malicious_df['Label'] = 'Malicious'
safe_df['Label'] = 'Normal'

# Combine datasets
combined_df = pd.concat([malicious_df, safe_df])

# Convert Timestamp column to datetime (update 'Timestamp' if necessary)
combined_df['Timestamp'] = pd.to_datetime(combined_df['Timestamp'])

# Sort data by Timestamp
combined_df.sort_values(by='Timestamp', inplace=True)

# Define explicit mappings for y-axis labels
label_mapping = {
    'Flow Duration': 'Flow Duration (seconds)',
    'Src Port': 'Source Port Number',
    'Dst Port': 'Destination Port Number',
    'Total Fwd Packets': 'Total Forward Packets',
    'Total Bwd Packets': 'Total Backward Packets',
    'Total Length of Fwd Packets': 'Forward Packets Length (Bytes)',
    'Total Length of Bwd Packets': 'Backward Packets Length (Bytes)',
    'Fwd Packet Length Max': 'Forward Packet Length Max (Bytes)',
    'Fwd Packet Length Min': 'Forward Packet Length Min (Bytes)',
    'Fwd Packet Length Mean': 'Forward Packet Length Mean (Bytes)',
    'Bwd Packet Length Max': 'Backward Packet Length Max (Bytes)',
    'Bwd Packet Length Min': 'Backward Packet Length Min (Bytes)',
    'Bwd Packet Length Mean': 'Backward Packet Length Mean (Bytes)',
    'Flow Bytes/s': 'Flow Bytes per second',
```

```
    'Flow Packets/s': 'Flow Packets per second',
    'Flow IAT Mean': 'Flow Interarrival Time Mean (ms)',
    # Add more mappings if necessary
}

# List features excluding Timestamp and Label
features_to_plot = combined_df.columns.drop(['Timestamp', 'Label'])

# Plotting with explicit axis labels
for feature in features_to_plot:
    plt.figure(figsize=(12, 6))

    # Malicious data
    malicious_data = combined_df[combined_df['Label'] == 'Malicious']
    plt.plot(malicious_data['Timestamp'], malicious_data[feature], color='red',
label='Malicious', alpha=0.7)

    # Normal data
    normal_data = combined_df[combined_df['Label'] == 'Normal']
    plt.plot(normal_data['Timestamp'], normal_data[feature], color='green', label='Normal',
alpha=0.7)

    # Explicit y-axis labeling
    ylabel = label_mapping.get(feature, feature)

    plt.title(f'{ylabel} over Time (Malicious vs Normal)')
    plt.xlabel('Timestamp')
    plt.ylabel(ylabel)

    plt.legend()
    plt.grid(alpha=0.5)
    plt.tight_layout()
    plt.show()
```

End-to-End Classification with Preprocessing and Confidence Filtering

A full machine learning pipeline is constructed to classify network traffic. The data is cleaned by dropping identifiers and IP-related fields. A `ColumnTransformer` is used to preprocess categorical and numerical features separately, applying clipping, imputation, and encoding. The model is trained using XGBoost, and both the pipeline and label encoder are saved for reuse.

Evaluation follows on a hold-out test set, with key metrics printed for analysis. Predictions are then made on a new dataset (`test.csv`). Each prediction is assigned a confidence score, and only those above a threshold of 0.65 are retained. These high-confidence results are labeled, quantified, and saved for further analysis.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer, LabelEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from xgboost import XGBClassifier
import joblib
import warnings

# Suppress XGBoost warnings
warnings.filterwarnings("ignore", category=UserWarning)

# Function to replace infinite values with NaN (replaces the lambda)
def replace_inf_with_nan(x):
    return np.where(np.isinf(x), np.nan, x)

# Load labeled training data
df = pd.read_csv('labeled_data.csv')

# Drop unwanted columns
X = df.drop(['Label', 'Flow ID', 'Timestamp', 'Src IP', 'Dst IP', 'Src Port', 'Dst Port'],
axis=1, errors='ignore')
y_raw = df['Label']

# Split data
X_train_raw, X_test_raw, y_train_raw, y_test_raw = train_test_split(
    X, y_raw, test_size=0.20, stratify=y_raw, random_state=42
)

# Encode string labels to numeric
label_encoder = LabelEncoder()
y_train = label_encoder.fit_transform(y_train_raw)
y_test = label_encoder.transform(y_test_raw)

# Set XGBoost objective
num_classes = len(label_encoder.classes_)
if num_classes == 2:
    objective = 'binary:logistic'
    eval_metric = 'logloss'
else:
    objective = 'multi:softprob'
    eval_metric = 'mlogloss'

# Define XGBoost parameters
xgb_params = {
    'n_estimators': 100,
    'max_depth': 10,
    'learning_rate': 0.1,
    'use_label_encoder': False,
```

```
    'eval_metric': eval_metric,
    'objective': objective,
    'random_state': 42,
    'n_jobs': -1
}
if num_classes > 2:
    xgb_params['num_class'] = num_classes

# Preprocessing pipeline
cat_cols = ['Protocol']
num_cols = [c for c in X_train_raw.columns if c not in cat_cols]
preprocessor = ColumnTransformer([
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols),
    ('num', Pipeline([
        ('clip', FunctionTransformer(replace_inf_with_nan, validate=False)),
        ('impute', SimpleImputer(strategy='median'))
    ]), num_cols)
])

# Build training pipeline
train_pipeline = Pipeline([
    ('pre', preprocessor),
    ('xgb', XGBClassifier(**xgb_params))
])

# Train the model
train_pipeline.fit(X_train_raw, y_train)

# Save the trained model
joblib.dump(train_pipeline, 'attack.pkl')
print("✅ Trained model saved to 'attack.pkl'")

# ——————————————————————————————————————————————
# Load model and predict on test.csv
# ——————————————————————————————————————————————
print("\n🔄 Loading model for predictions...")
loaded_pipeline = joblib.load('attack.pkl')

# Evaluate loaded model on hold-out test set
y_pred = loaded_pipeline.predict(X_test_raw)
print("\n📊 Evaluation on Test Set:")
print("Number of features used for training:",
loaded_pipeline.named_steps['pre'].transform(X_train_raw).shape[1])
print("Classification Report:\n", classification_report(y_test, y_pred,
target_names=label_encoder.classes_))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))

# Load test data
test_df = pd.read_csv('test.csv')
test_features = test_df.drop(['Flow ID', 'Timestamp', 'Src IP', 'Dst IP', 'Src Port', 'Dst
Port', 'Label'], axis=1, errors='ignore')
```

```
# Predict using loaded model
test_pred_numeric = loaded_pipeline.predict(test_features)
test_pred_label = label_encoder.inverse_transform(test_pred_numeric)
test_proba = loaded_pipeline.predict_proba(test_features).max(axis=1)

# Add predictions to DataFrame
test_df['Predicted_Label'] = test_pred_label
test_df['Confidence'] = test_proba

# Filter low-confidence predictions
filtered_df = test_df[test_df['Confidence'] >= 0.65]

# Summary
print("\n📈 Prediction Summary (Confidence ≥ 0.65):")
print(filtered_df['Predicted_Label'].value_counts())
print("\nPrediction Proportions (%):")
print((filtered_df['Predicted_Label'].value_counts(normalize=True) * 100).round(2))

# Save predictions
filtered_df.to_csv('other_data_labeled.csv', index=False)
print(f"\n✅ Filtered predictions saved to 'other_data_labeled.csv' (kept {len(filtered_df)}
of {len(test_df)} rows)")
```

Re-using the Trained Model for Inference and Reporting

This block focuses on inference using the previously trained pipeline. The saved model is loaded and evaluated on the test set. Standard classification metrics—including accuracy, confusion matrix, and class-wise performance—are computed and printed.

New predictions are then generated for an external dataset. Confidence scores accompany each prediction, and only rows with confidence ≥ 0.65 are kept. Final results are saved to a labeled output file and summarized by count and proportion to offer a quick assessment of the predicted distribution.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import joblib
# ——————————————————————————————————————————
# Load model and predict on test.csv
# ——————————————————————————————————————————
print("\n🔄 Loading model for predictions...")
loaded_pipeline = joblib.load('attack.pkl')
```

```
# Evaluate loaded model on hold-out test set
y_pred = loaded_pipeline.predict(X_test_raw)
print("\n📊 Evaluation on Test Set:")
print("Number of features used for training:",
loaded_pipeline.named_steps['pre'].transform(X_train_raw).shape[1])
print("Classification Report:\n", classification_report(y_test, y_pred,
target_names=label_encoder.classes_))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))

# Load test data
test_df = pd.read_csv('test.csv')
test_features = test_df.drop(['Flow ID', 'Timestamp', 'Src IP', 'Dst IP', 'Src Port', 'Dst
Port', 'Label'], axis=1, errors='ignore')

# Predict using loaded model
test_pred_numeric = loaded_pipeline.predict(test_features)
test_pred_label = label_encoder.inverse_transform(test_pred_numeric)
test_proba = loaded_pipeline.predict_proba(test_features).max(axis=1)

# Add predictions to DataFrame
test_df['Predicted_Label'] = test_pred_label
test_df['Confidence'] = test_proba

# Filter low-confidence predictions
filtered_df = test_df[test_df['Confidence'] >= 0.65]

# Summary
print("\n📈 Prediction Summary (Confidence ≥ 0.65):")
print(filtered_df['Predicted_Label'].value_counts())
print("\nPrediction Proportions (%):")
print((filtered_df['Predicted_Label'].value_counts(normalize=True) * 100).round(2))

# Save predictions
filtered_df.to_csv('other_data_labeled.csv', index=False)
print(f"\n✅ Filtered predictions saved to 'other_data_labeled.csv' (kept {len(filtered_df)}
of {len(test_df)} rows)")
```

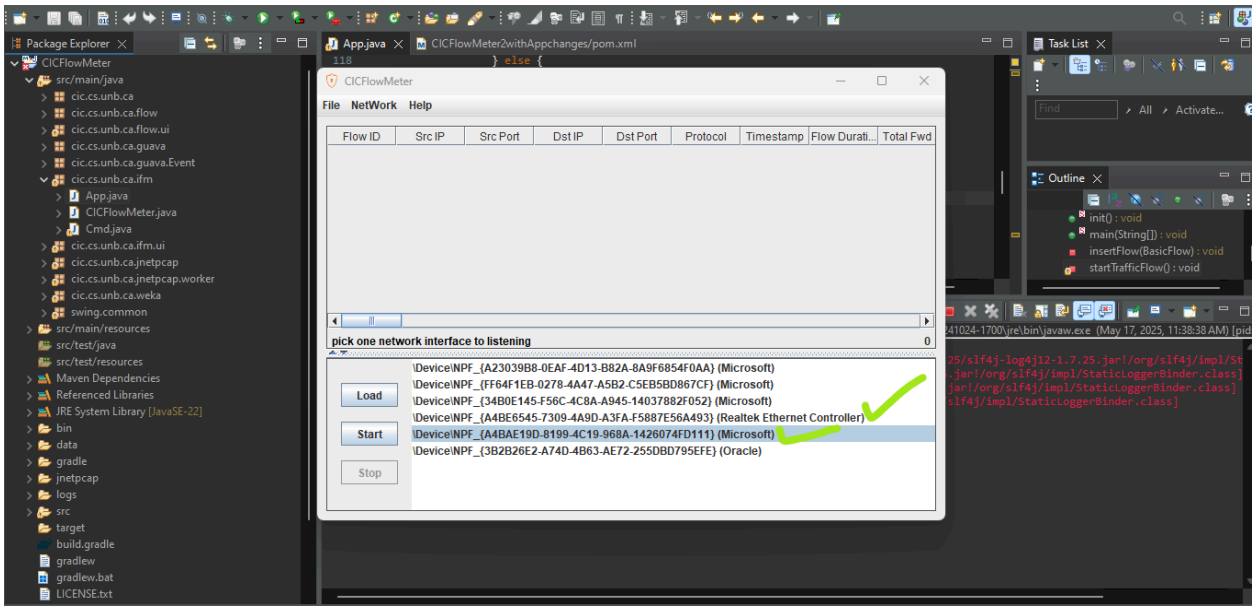
Integration

Check What Inteface You Are Using

Firstly, let's find out the interface that we are using to capture packets in CICFlowMeter

Open Eclipse, and note the interface that you are using to capture packets.

For example,



Here, I'm using the interface \\Device\\NPF_{A4BAE19D-8199-4C19-968A-1426074FD111} for connecting wirelessly to wifi.
If I would use ethernet, then I would select interface \\Device\\NPF_{A4BE6545-7309-4A9D-A3FA-F5887E56A493} instead.

Now, we'll modify the app.java code and the pom.xml code, so that it pushes these packets to the kafka topic and kafka server, so that the python script in pycharm can read the packets.

Modify App.java and Pom.xml

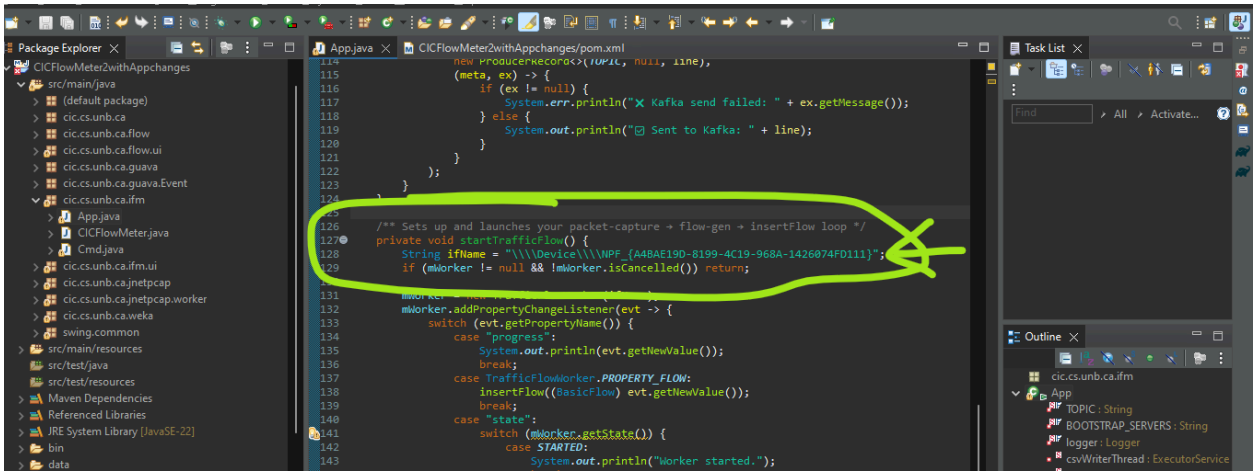
So in CICFlowMeter, modify app.java to this:

```
package cic.cs.unb.ca.ifm;

import cic.cs.unb.ca.flow.FlowMgr;
import cic.cs.unb.ca.guava.GuavaMgr;
import cic.cs.unb.ca.jnetpcap.BasicFlow;
import cic.cs.unb.ca.jnetpcap.FlowFeature;
import cic.cs.unb.ca.jnetpcap.worker.InsertCsvRow;
import cic.cs.unb.ca.jnetpcap.worker.TrafficFlowWorker;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import javax.swing.*;
import java.awt.EventQueue;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
```

```
import java.util.concurrent.CancellationException;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.logging.FileHandler;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;

public class App {

    private static final String TOPIC             = "CicFlowmeter";
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";
    private static final Logger logger = Logger.getLogger(App.class.getName());

    // single-threaded CSV writer
    private static ExecutorService csvWriterThread;
    // single, long-lived Kafka producer
    private static Producer<String,String> kafkaProducer;

    /** Initialize FlowMgr, GuavaMgr, CSV executor and the one KafkaProducer */
    public static void init() {
        // 1) CSV writer
        csvWriterThread = Executors.newSingleThreadExecutor();

        // 2) FlowMgr & GuavaMgr
        FlowMgr.getInstance().init();
        GuavaMgr.getInstance().init();

        // 3) ONE shared KafkaProducer
        Properties props = new Properties();
        props.put("bootstrap.servers",      BOOTSTRAP_SERVERS);
        props.put("acks",                   "all");
        props.put("enable.idempotence",     "true");
        props.put("key.serializer",         "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",       "org.apache.kafka.common.serialization.StringSerializer");
        kafkaProducer = new KafkaProducer<>(props);

        // 4) JVM shutdown hook to close resources cleanly
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("Shutting down CSV writer and Kafka producer...");
            csvWriterThread.shutdownNow();
            kafkaProducer.flush();
            kafkaProducer.close();
            System.out.println("Shutdown complete.");
        }));
    }

    public static void main(String[] args) {
        // configure file logging
        try {
            FileHandler fh = new FileHandler("features.log");
            fh.setFormatter(new SimpleFormatter());
            logger.addHandler(fh);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // start the Swing capture worker on the EDT
        EventQueue.invokeLater(() -> {
            try {
                init();
                GuavaMgr.getInstance().getEventBus().register(App.class);
                App app = new App();
                app.startTrafficFlow();
            } catch (Exception e) {
                System.err.println("Startup error: " + e.getMessage());
            }
        });

        // keep the JVM alive without busy-spinning
```

```
        while (true) {
            try {
                Thread.sleep(60_000);
            } catch (InterruptedException ignored) {
            }
        }
    }

    /** Called by TrafficFlowWorker whenever a new flow arrives */
    private void insertFlow(BasicFlow flow) {
        // 1) Write to CSV
        List<String> rows = new ArrayList<>();
        rows.add(flow.dumpFlowBasedFeaturesEx());
        String header   = FlowFeature.getHeader();
        String path     = FlowMgr.getInstance().getSavePath();
        String filename = LocalDate.now() + FlowMgr.FLOW_SUFFIX;
        csvWriterThread.execute(new InsertCsvRow(header, rows, path, filename));

        // 2) Local file log
        logger.info(rows.get(0));

        // 3) Send to Kafka via the single producer
        for (String line : rows) {
            kafkaProducer.send(
                new ProducerRecord<>(TOPIC, null, line),
                (meta, ex) -> {
                    if (ex != null) {
                        System.err.println("❌ Kafka send failed: " + ex.getMessage());
                    } else {
                        System.out.println("✅ Sent to Kafka: " + line);
                    }
                }
            );
        }
    }

    /** Sets up and launches your packet-capture → flow-gen → insertFlow loop */
    private void startTrafficFlow() {
        String ifName = "\\\\Device\\\\NPF_{A4BAE19D-8199-4C19-968A-1426074FD111}";
        if (mWorker != null && !mWorker.isCancelled()) return;

        mWorker = new TrafficFlowWorker(ifName);
        mWorker.addPropertyChangeListener(evt -> {
            switch (evt.getPropertyName()) {
                case "progress":
                    System.out.println(evt.getNewValue());
                    break;
                case TrafficFlowWorker.PROPERTY_FLOW:
                    insertFlow((BasicFlow) evt.getNewValue());
                    break;
                case "state":
                    switch (mWorker.getState()) {
                        case STARTED:
                            System.out.println("Worker started.");
                            break;
                        case DONE:
                            try {
                                System.out.println(mWorker.get());
                            } catch (CancellationException|InterruptedException|ExecutionException ex) {
                                System.err.println("Worker error: " + ex.getMessage());
                            }
                            break;
                    }
                    break;
            }
        });
        mWorker.execute();
    }

    private TrafficFlowWorker mWorker;
```

}



You need to change your ifname value according to what your interface is, that we have noted above when we checked what interface we ourself are using. Each laptop or machine may have its own interface. For this laptop, the interface values mentioned in the code may not work for your laptop or machine, so as mentioned above, we first have to note the interface that our machine is using for either wireless wifi, or ethernet.

Now modify pom.xml to this:

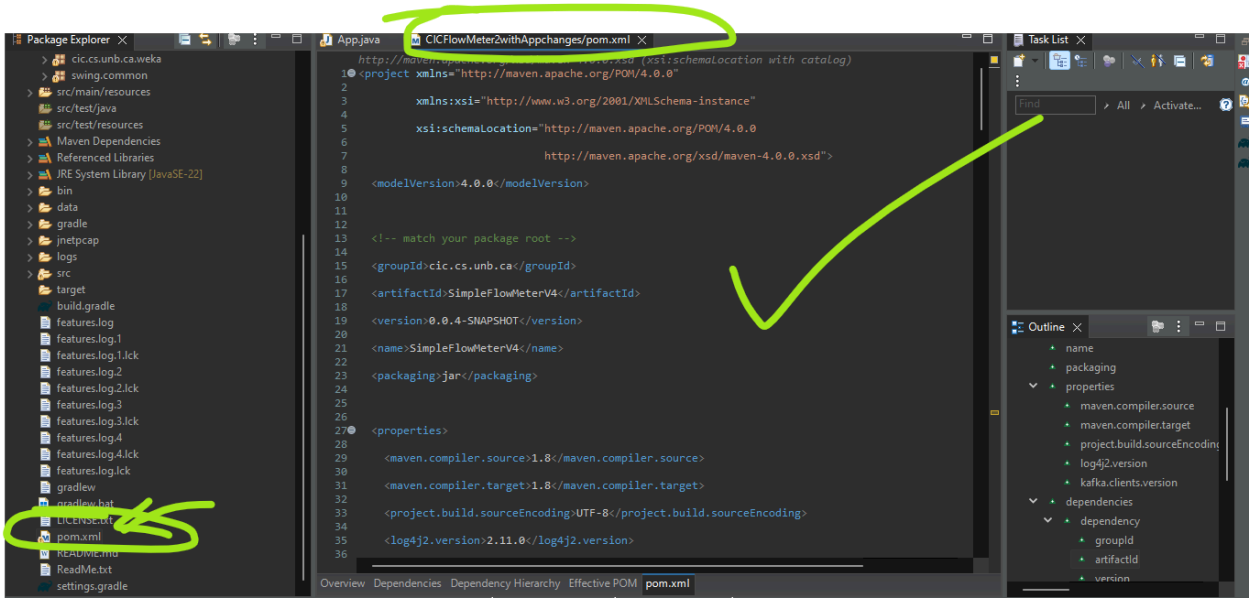
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                            http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- match your package root -->
  <groupId>cic.cs.unb.ca</groupId>
  <artifactId>SimpleFlowMeterV4</artifactId>
  <version>0.0.4-SNAPSHOT</version>
  <name>SimpleFlowMeterV4</name>
  <packaging>jar</packaging>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
```

```
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <log4j2.version>2.11.0</log4j2.version>
    <kafka.clients.version>3.5.1</kafka.clients.version>
  </properties>

  <dependencies>
    <!-- Kafka Clients -->
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>${kafka.clients.version}</version>
    </dependency>

    <!-- Log4j2 + SLF4J binding -->
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>${log4j2.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>${log4j2.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-slf4j-impl</artifactId>
      <version>${log4j2.version}</version>
    </dependency>

    <!-- JNetPcap (native libs must be on your LD_LIBRARY_PATH) -->
    <dependency>
      <groupId>org.jnetpcap</groupId>
      <artifactId>jnetpcap</artifactId>
```

```
    <version>1.4.1</version>
  </dependency>

  <!-- Testing -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>

  <!-- Commons, Weka, Tika, etc. -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.6</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.5</version>
  </dependency>
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.5</version>
  </dependency>
  <dependency>
    <groupId>org.jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.19</version>
  </dependency>
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
```

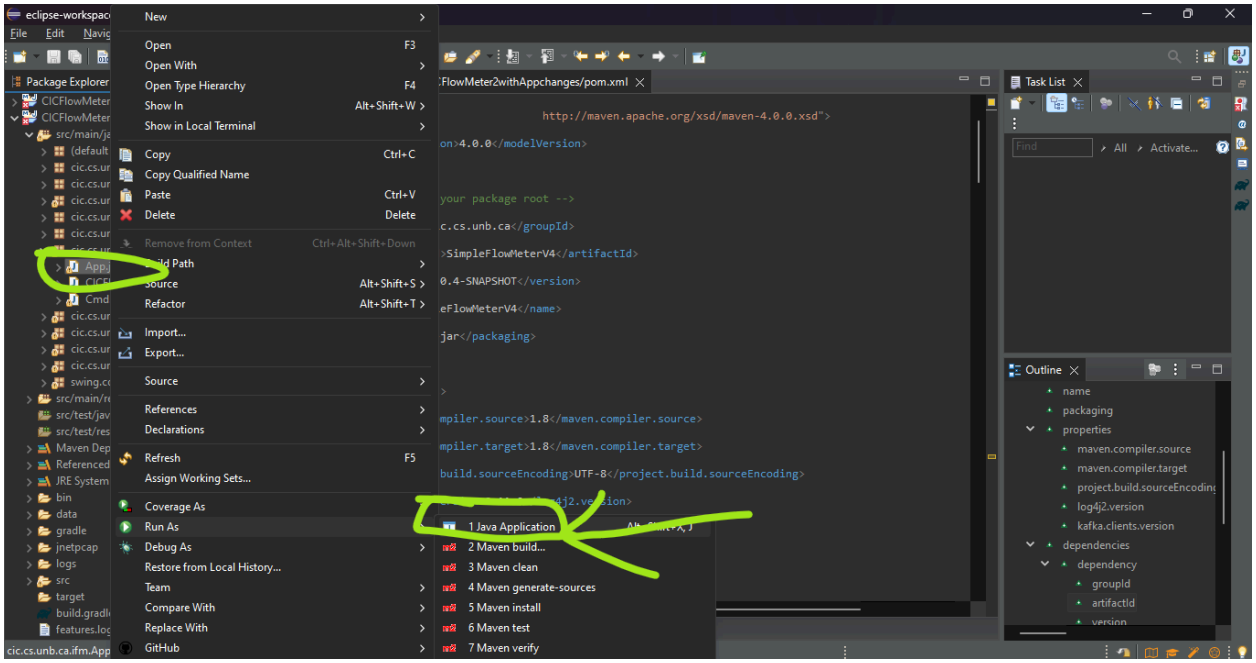
```
    <version>23.6-jre</version>
  </dependency>
  <dependency>
    <groupId>nz.ac.waikato.cms.weka</groupId>
    <artifactId>weka-stable</artifactId>
    <version>3.6.14</version>
  </dependency>
  <dependency>
    <groupId>org.apache.tika</groupId>
    <artifactId>tika-core</artifactId>
    <version>1.17</version>
  </dependency>
 </dependencies>

 <build>
  <plugins>
    <!-- Keep your excludes & Java version settings -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>**/*_bak.java</exclude>
          <exclude>**/OnLineFlowMeter.java</exclude>
        </excludes>
        <source>${maven.compiler.source}</source>
        <target>${maven.compiler.target}</target>
        <encoding>${project.build.sourceEncoding}</encoding>
      </configuration>
    </plugin>

    <!-- Shade all deps into one “uber-jar” -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
```

```
      <version>3.2.4</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals><goal>shade</goal></goals>
          <configuration>
            <!-- no relocation needed unless you have conflicts -->
            <createDependencyReducedPom>false</createDependencyReducedPom>
            <transformers>
              <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>cic.cs.unb.ca.ifm.App</mainClass>
              </transformer>
            </transformers>
            <finalName>${project.artifactId}-${project.version}</finalName>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
 </build>
</project>
```

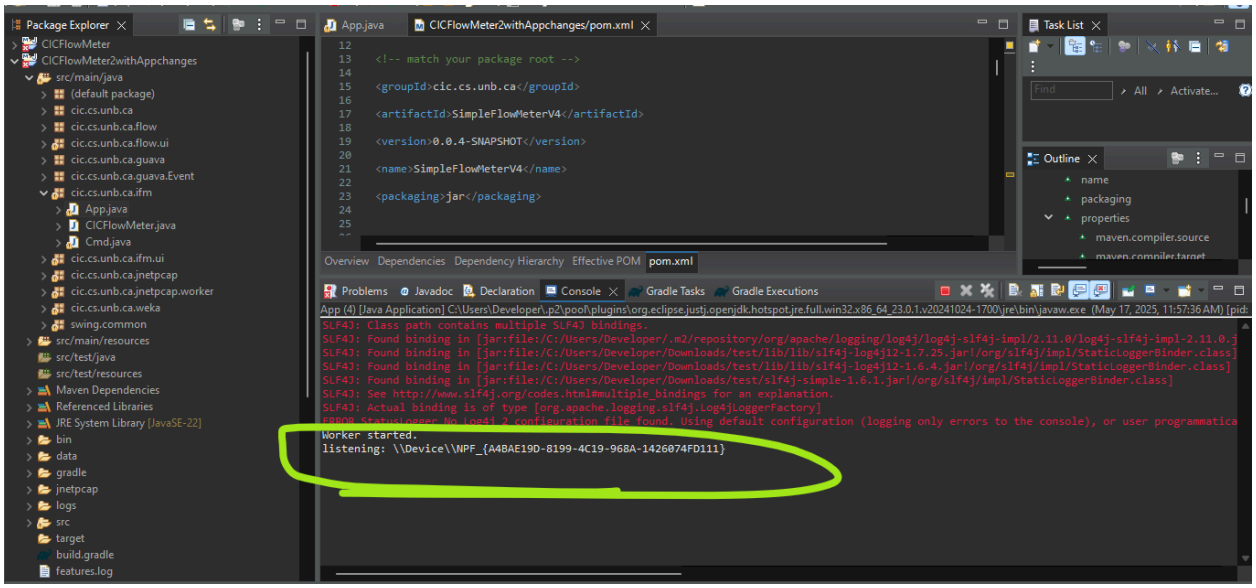


Run the modified CICFlowMeter

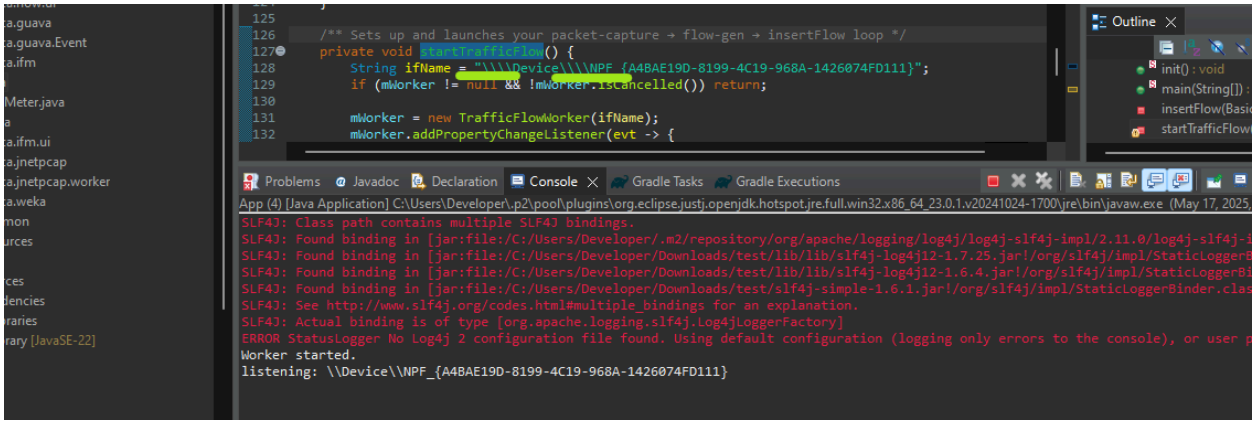
Now run the app.java



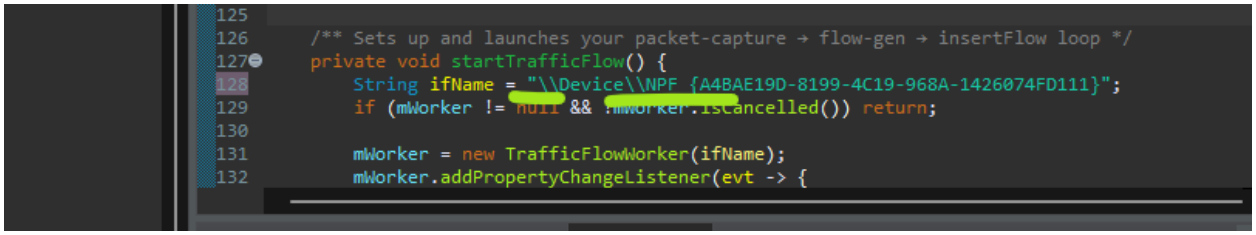
It should show something like this:



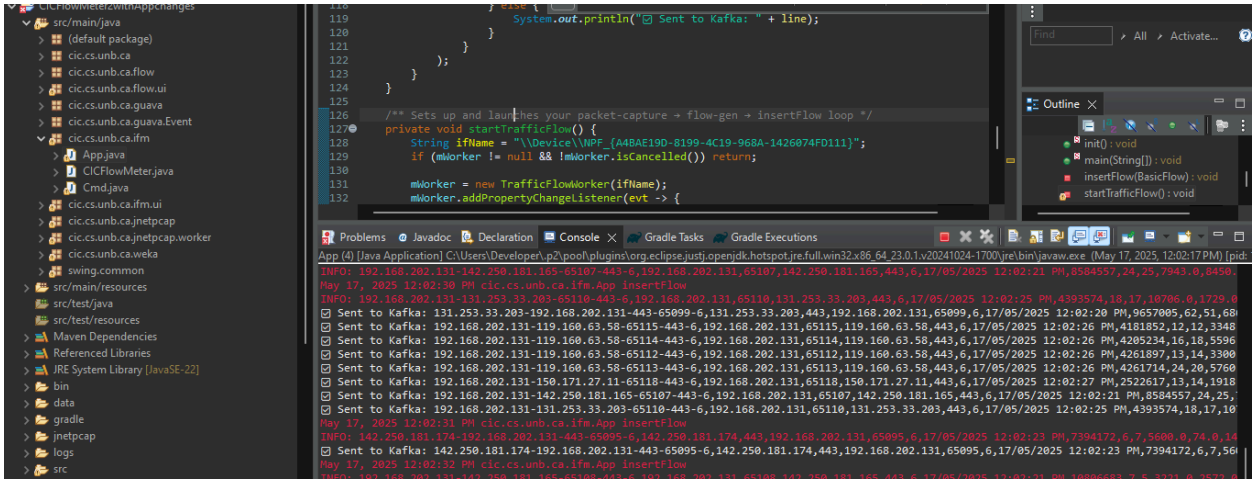
Now the packets should start showing on the console instantly, so go ahead and search something on the internet, but if the packets dont show up in under a minute, then there's an issue. Try something like changing the four "\\\\" into two "\\", so currently it's like this:



There's 4 backward slashes, let's make that two.



Hit ctrl+s to save, and now let's run the app.java again.

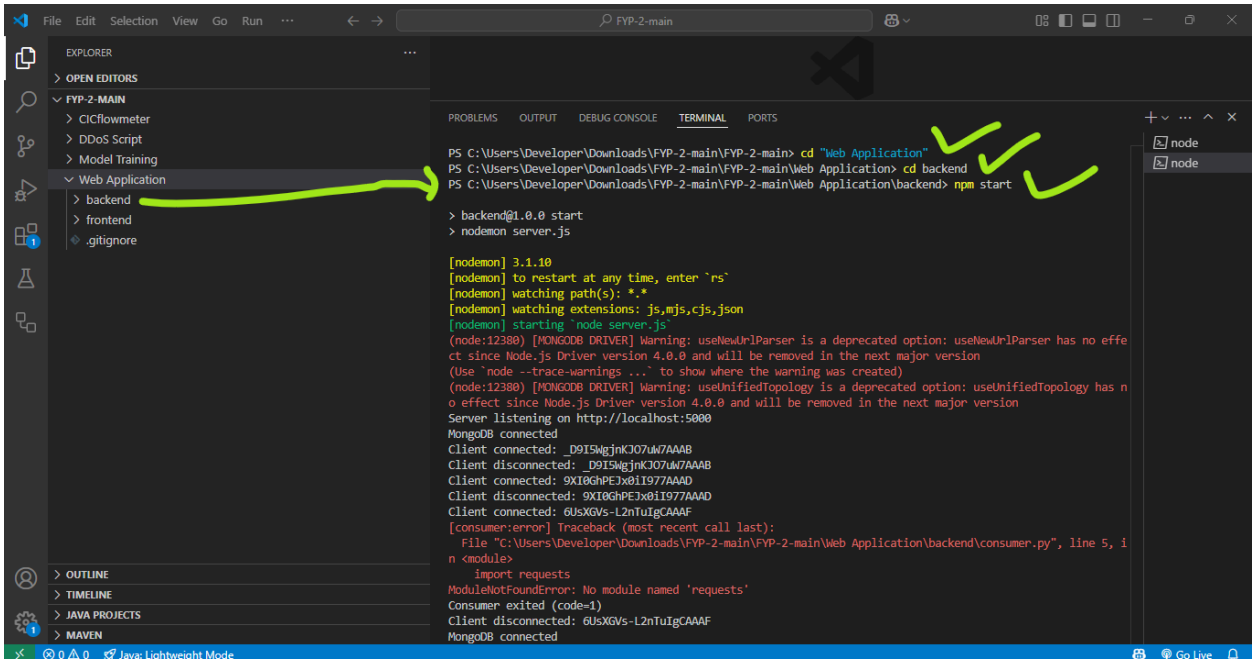


As you can see it started capturing packets, but don't worry if that doesn't work and you still don't see any packets being captured, then simply change the two backward slashes into four backward slashes just like it was before, and then run the app again, and then it should start capturing packets.

Setup and launch Front End and Backend of Website

Download the website code from here: https://github.com/Ahmad-Imran5374/FYP-2

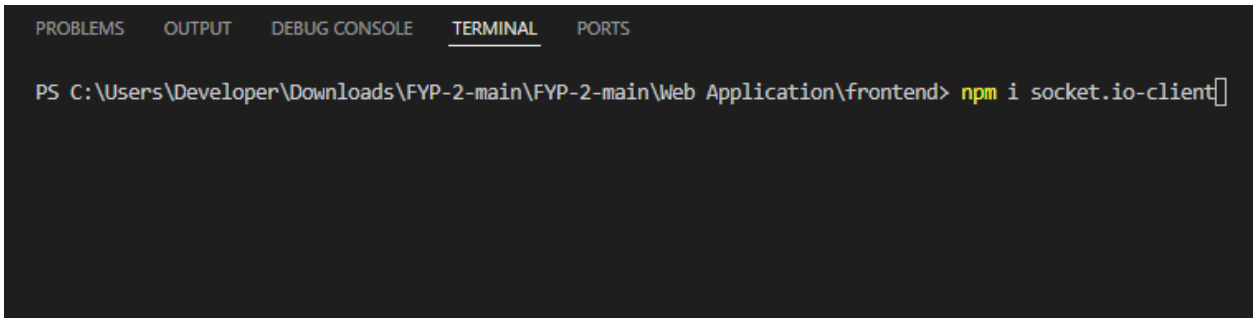
Launch Backend



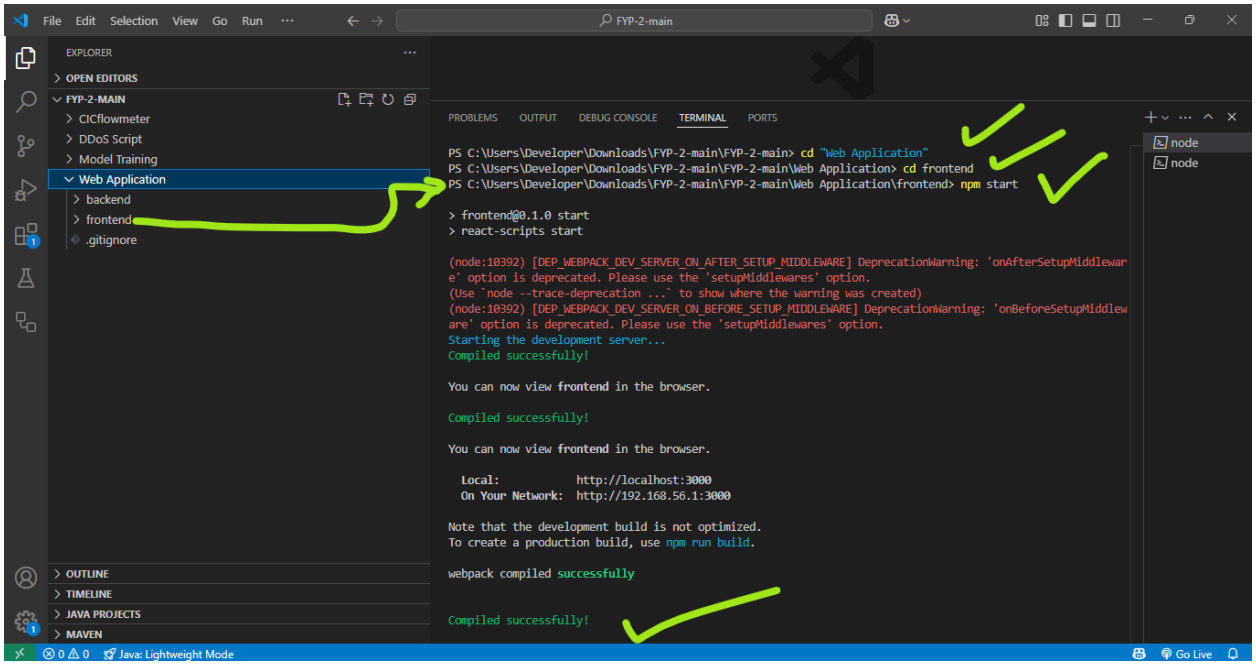
Extract the website's zip file you downloaded from github, open the folder in Vscode, and open a powershell terminal inside VsCode and navigate to backend folder and type **npm start** and press enter inside the backend folder.

Launch Frontend

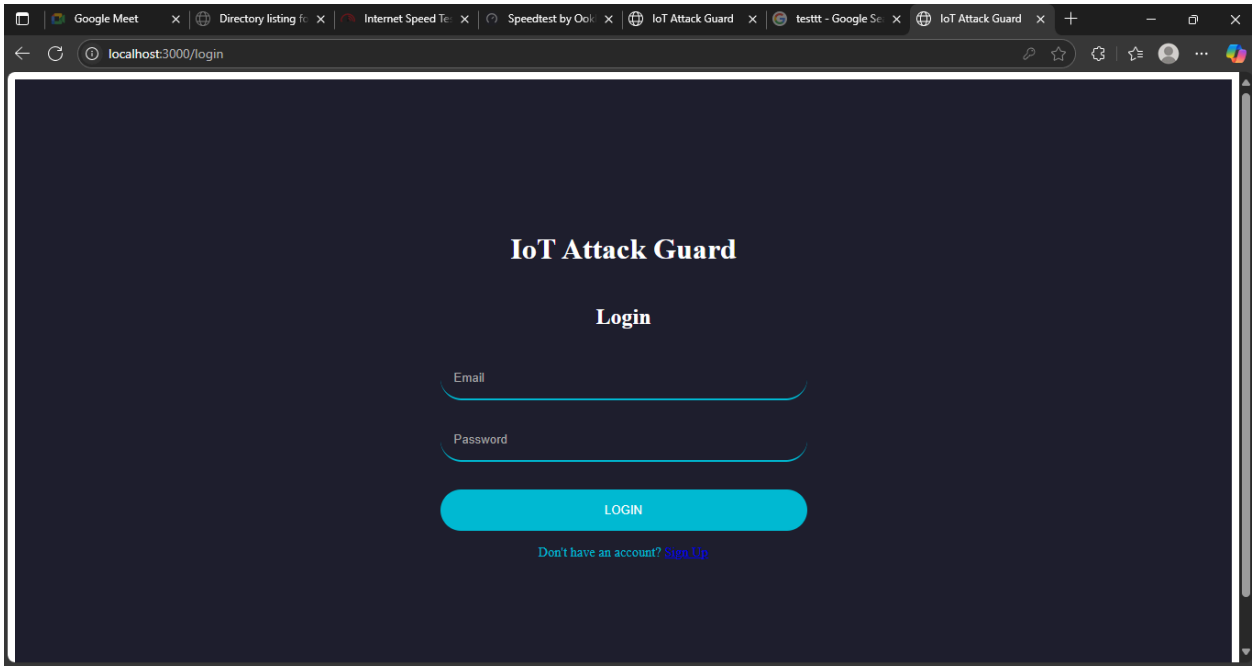
Now navigate to the frontend folder and type **npm i socket.io-client**



Once that is done, type **npm start** and press enter inside the frontend folder as shown in the pic below.



Now open the localhost in the browser:

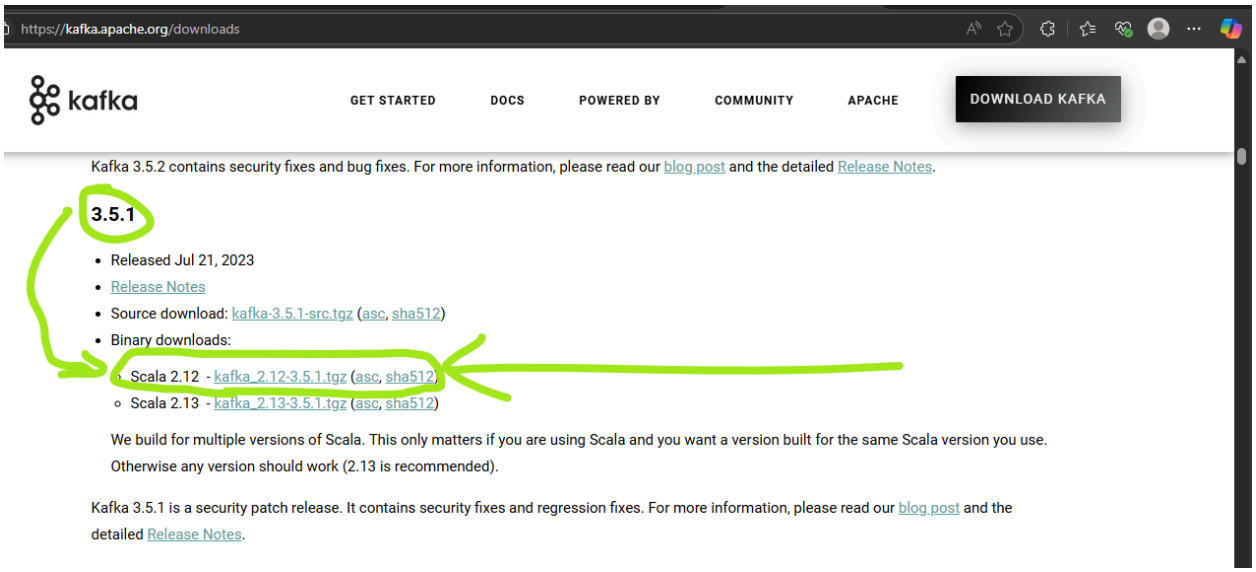


Kafka Setup

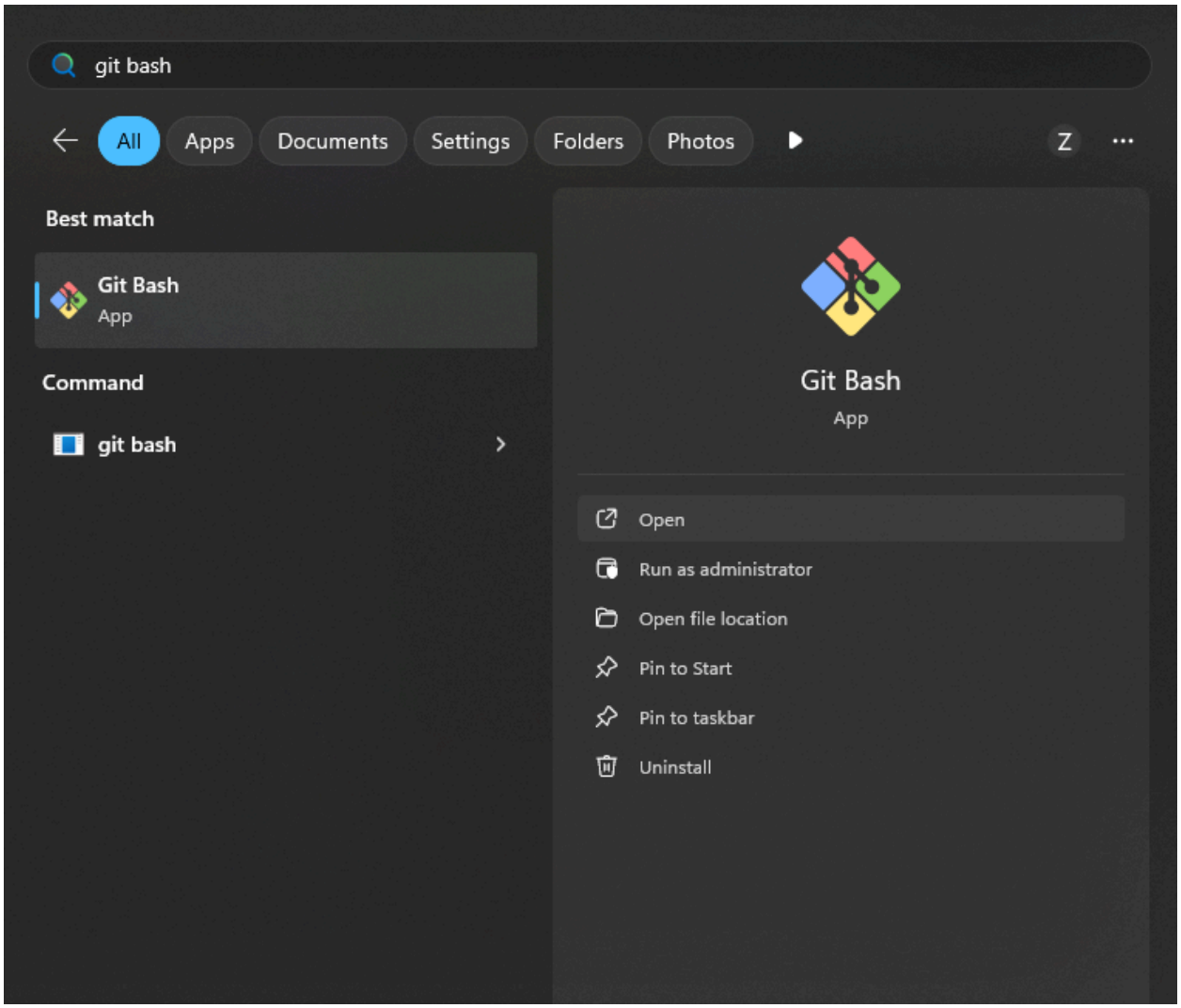
Create Kafka Topic

Download Git Bash from https://git-scm.com/downloads and open it.

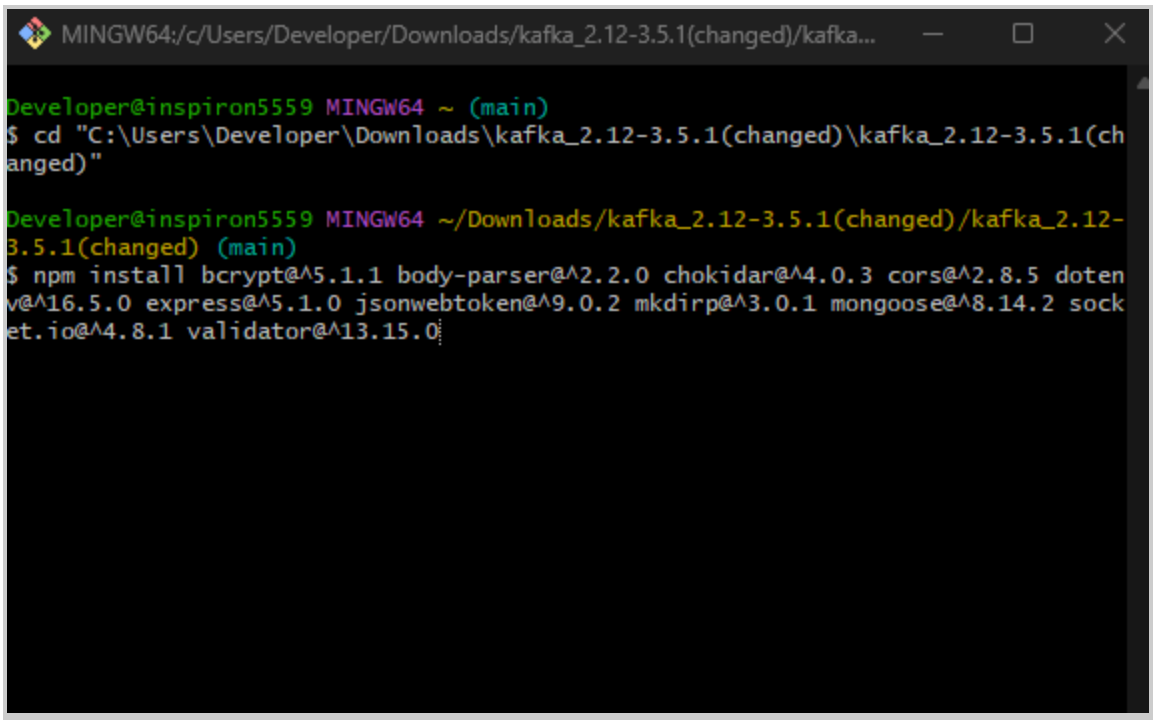
Now download kafka 2.12-3.5.1 from https://kafka.apache.org/downloads



Extract the kafka_2.12-3.5.1.tgz file and open git bash.

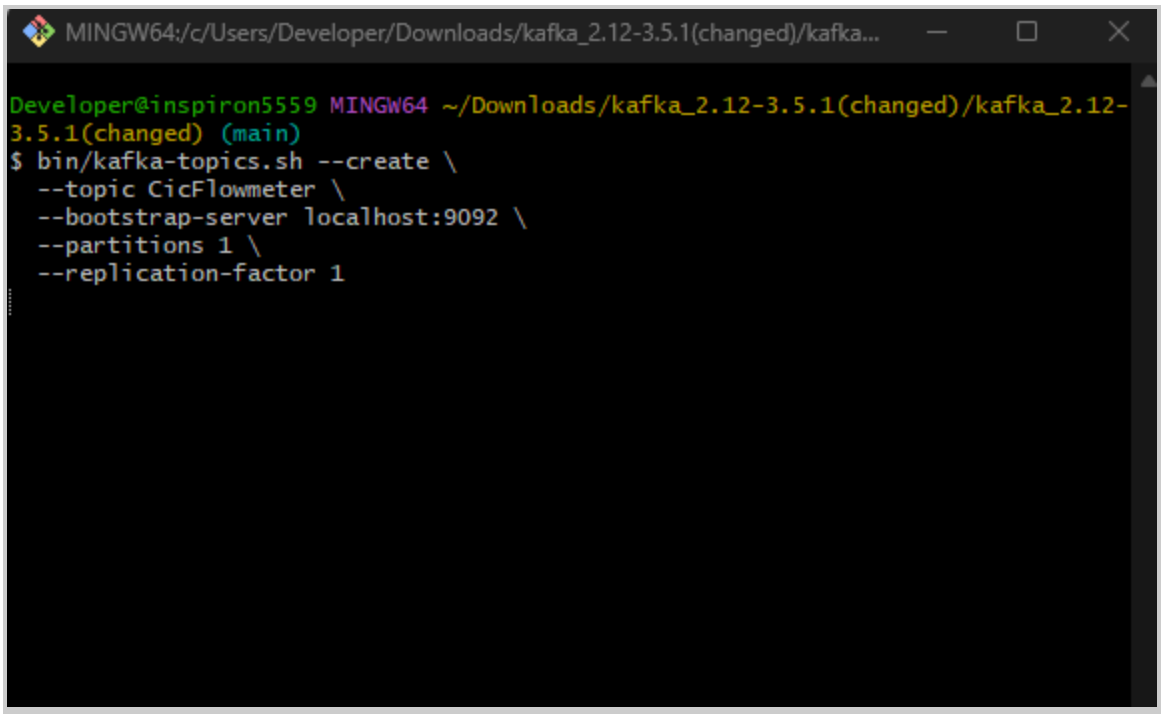


Navigate to the extracted kafka folder and type this command to install packages:
npm install bcrypt@^5.1.1 body-parser@^2.2.0 chokidar@^4.0.3 cors@^2.8.5 dotenv@^16.5.0 express@^5.1.0 jsonwebtoken@^9.0.2 mkdirp@^3.0.1 mongoose@^8.14.2 socket.io@^4.8.1 validator@^13.15.0



Now type this command to create the kafka **topic**:

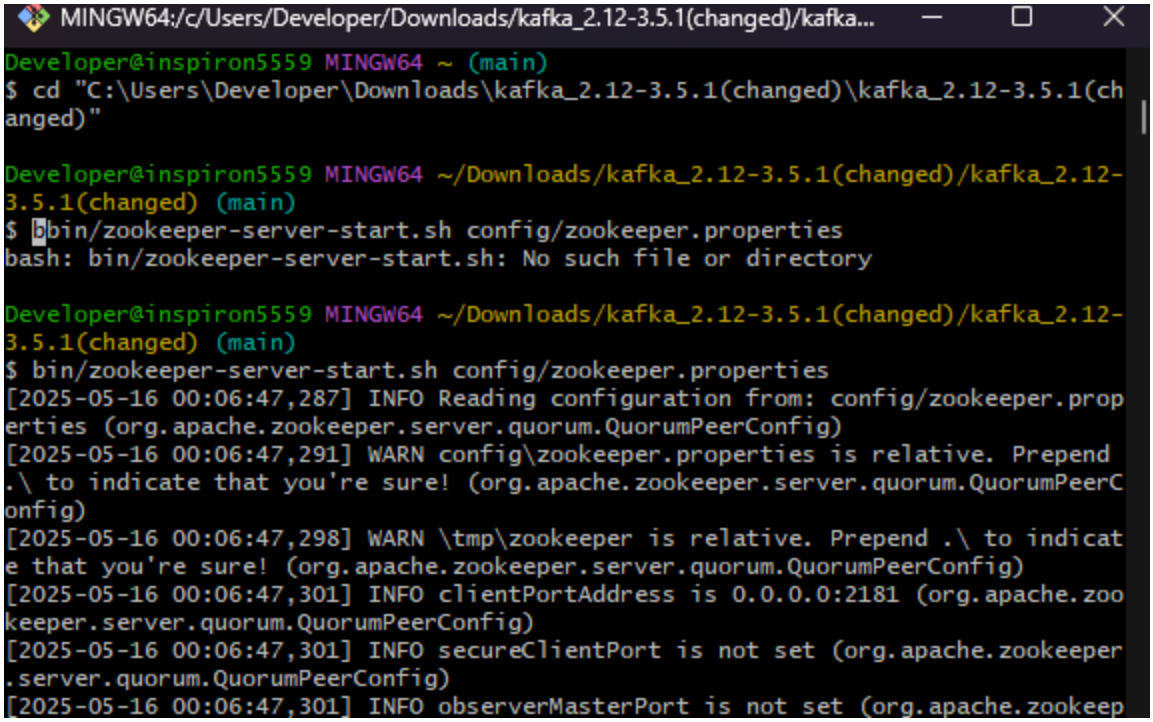
```
bin/kafka-topics.sh --create \
  --topic CicFlowmeter \
  --bootstrap-server localhost:9092 \
  --partitions 1 \
  --replication-factor 1
```



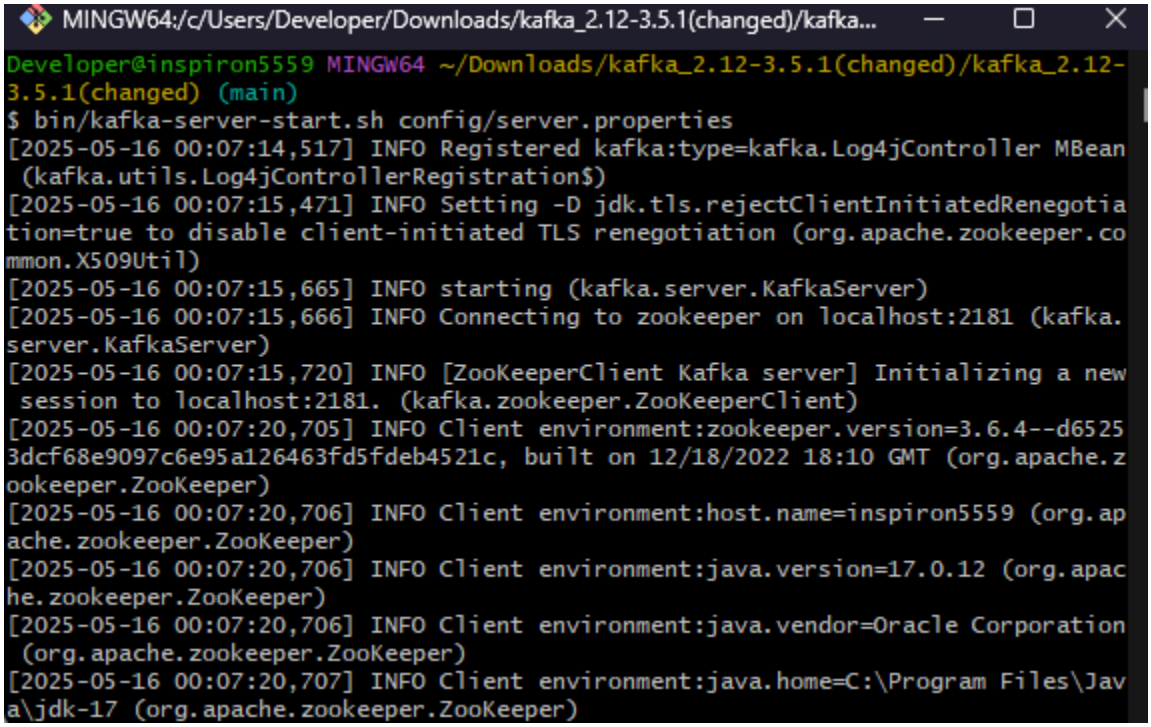
Now you can close this window.

Start Kafka Server

Open two git bash windows now. In the first window, navigate to the kafka folder and type `bin/zookeeper-server-start.sh config/zookeeper.properties`



In the second window, navigate to the kafka folder and type:
bin/kafka-server-start.sh config/server.properties

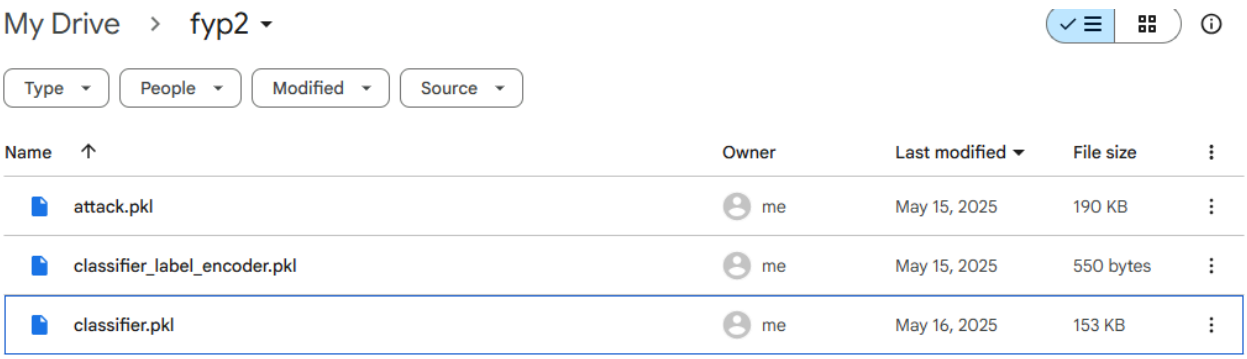


Python script setup

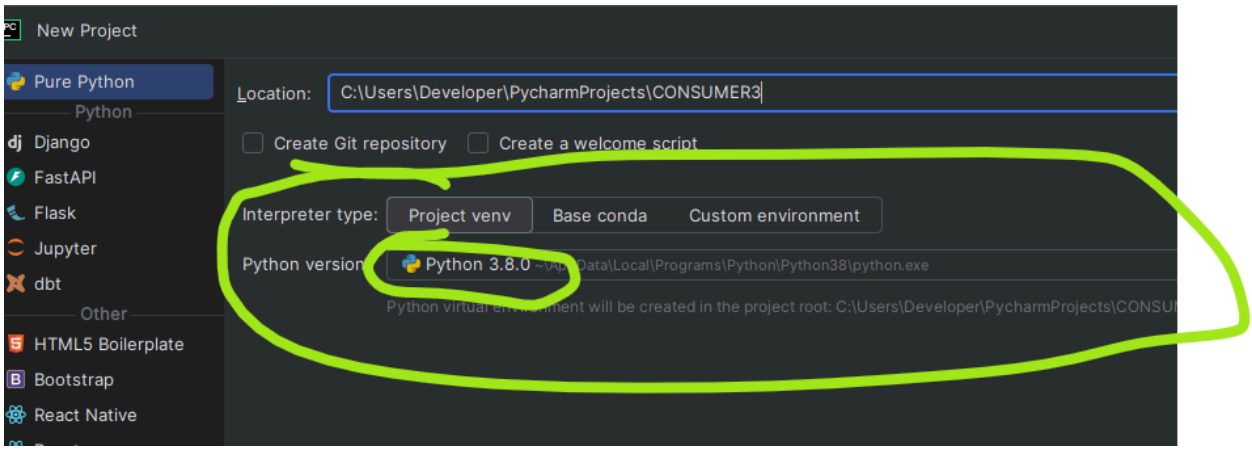
Download Python **3.8.0** from https://www.python.org/downloads/

We must use python 3.8.0 for now to avoid any issues. Otherwise there can be issues.

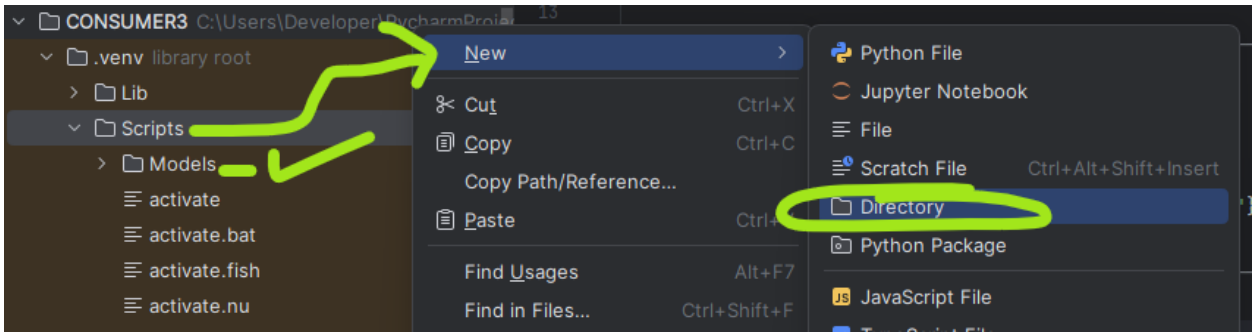
Download Pycharm from https://www.jetbrains.com/pycharm/
Download the three files for our models from
https://drive.google.com/drive/folders/1sla35ybd-XUdZdHnJ_of7B0wNdlBLfPE



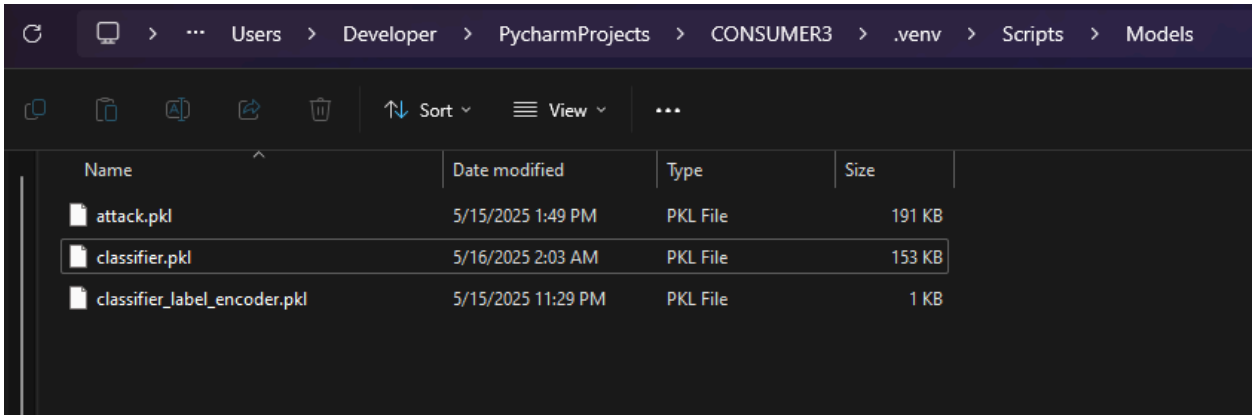
Create a project in Pycharm and name it **CONSUMER3** and make sure to set the Python interpreter version set to Python 3.8.0



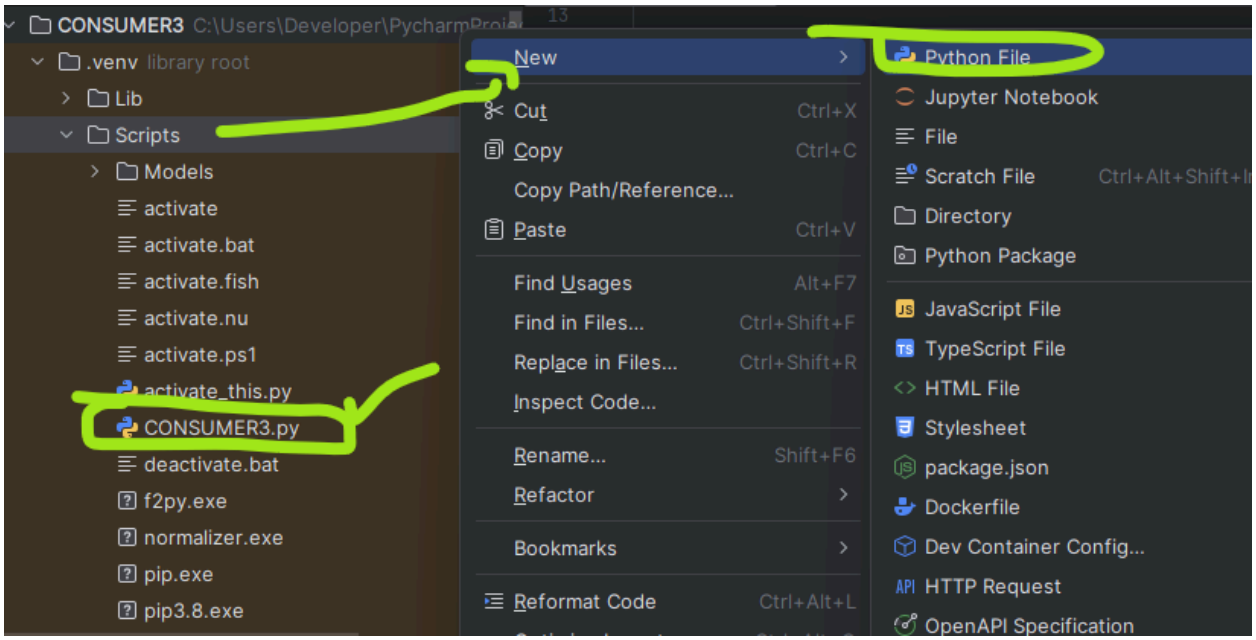
Then inside the .venv/Scripts make a new **Directory** and name it **Models**



Now navigate to the Models folder from your file explorer, and place the three downloaded model files inside.



Now create a new Python file inside .venv/Scripts and call name it **CONSUMER3**



Paste this code inside the CONSUMER3.py

```python
import __main__
import joblib
import numpy as np
import pandas as pd
from kafka import KafkaConsumer
import requests
from requests.exceptions import RequestException

# Monkey patch for loading custom transformers
def replace_inf_with_nan(x):
    return np.where(np.isinf(x), np.nan, x)
__main__.replace_inf_with_nan = replace_inf_with_nan

# ─────────────────────────────────────────────────────
# CORRECTED LABEL MEANINGS
MALICIOUS_LABEL = 0  # malicious = 0
NORMAL_LABEL = 1     # normal = 1
LABEL_MAP = {0: "Malicious", 1: "Normal"}
DEVICE_LABELS = {0: "IoT Device", 1: "Non-IoT Device"}

# ─────────────────────────────────────────────────────
# CONFIGURATION
MODEL1_CONF_THRESHOLD = 0.50
MODEL2_CONF_THRESHOLD = 0.95
MERN_ENDPOINT = "http://localhost:5000/api/packets"
KAFKA_TOPIC = "CicFlowmeter"
KAFKA_BOOTSTRAP_SERVERS = ["localhost:9092"]

# Flow feature columns used during classifier training
FEATURE_COLUMNS = [
    'Protocol', 'Flow Duration', 'Total Fwd Packet', 'Total Bwd packets',
    'Total Length of Fwd Packet', 'Total Length of Bwd Packet',
    'Fwd Packet Length Max', 'Fwd Packet Length Min', 'Fwd Packet Length Mean',
    'Fwd Packet Length Std', 'Bwd Packet Length Max', 'Bwd Packet Length Min',
    'Bwd Packet Length Mean', 'Bwd Packet Length Std', 'Flow Bytes/s',
    'Flow Packets/s', 'Flow IAT Mean', 'Flow IAT Std', 'Flow IAT Max',
    'Flow IAT Min', 'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std',
    'Fwd IAT Max', 'Fwd IAT Min', 'Bwd IAT Total', 'Bwd IAT Mean',
    'Bwd IAT Std', 'Bwd IAT Max', 'Bwd IAT Min', 'Fwd PSH Flags',
    'Bwd PSH Flags', 'Fwd URG Flags', 'Bwd URG Flags', 'Fwd Header Length',
    'Bwd Header Length', 'Fwd Packets/s', 'Bwd Packets/s', 'Packet Length Min',
    'Packet Length Max', 'Packet Length Mean', 'Packet Length Std',
    'Packet Length Variance', 'FIN Flag Count', 'SYN Flag Count',
    'RST Flag Count', 'PSH Flag Count', 'ACK Flag Count', 'URG Flag Count',
    'CWR Flag Count', 'ECE Flag Count', 'Down/Up Ratio', 'Average Packet Size',
    'Fwd Segment Size Avg', 'Bwd Segment Size Avg', 'Fwd Bytes/Bulk Avg',
    'Fwd Packet/Bulk Avg', 'Fwd Bulk Rate Avg', 'Bwd Bytes/Bulk Avg',
    'Bwd Packet/Bulk Avg', 'Bwd Bulk Rate Avg', 'Subflow Fwd Packets',
    'Subflow Fwd Bytes', 'Subflow Bwd Packets', 'Subflow Bwd Bytes',
```

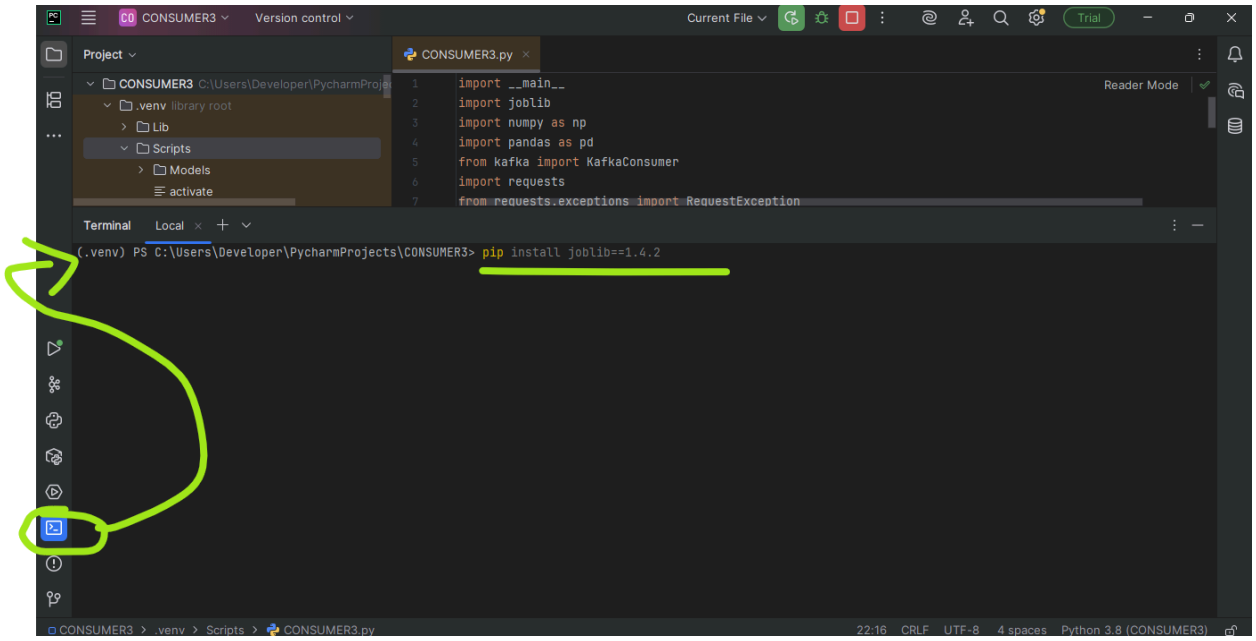
```
    'FWD Init Win Bytes', 'Bwd Init Win Bytes', 'Fwd Act Data Pkts',
    'Fwd Seg Size Min', 'Active Mean', 'Active Std', 'Active Max',
    'Active Min', 'Idle Mean', 'Idle Std', 'Idle Max', 'Idle Min'
]

# ——————————————————————————————————————————————
# Load trained models
classifier_pipeline = joblib.load("Models/classifier.pkl")
attack_model = joblib.load("Models/attack.pkl")

# ——————————————————————————————————————————————
# Setup Kafka Consumer
consumer = KafkaConsumer(
    KAFKA_TOPIC,
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    auto_offset_reset="earliest",
    enable_auto_commit=True,
    group_id="flow-classifier",
    value_deserializer=lambda m: m.decode("utf-8")
)
print("[✓] Kafka consumer listening for messages...")

# ——————————————————————————————————————————————
streak = {'malicious': 0}

for msg in consumer:
    raw = msg.value.strip()
    print(f"\n[Raw Kafka Message] {raw}")

    try:
        parts = raw.split(",")
        while parts and parts[-1] in ("", "NeedManualLabel"):
            parts.pop()

        if len(parts) < 7:
            print("[!] Skipping: not enough fields")
            continue

        flow_id, src_ip, src_port, dst_ip, dst_port, protocol, timestamp =
parts[:7]
        features = [float(x) for x in parts[7:]]

        if len(features) != len(FEATURE_COLUMNS) - 1:
            print(f"[!] Feature count mismatch: {len(features)} found vs
{len(FEATURE_COLUMNS) - 1} expected")
            continue

        # Build DataFrame
        df = pd.DataFrame([features], columns=FEATURE_COLUMNS[1:])
```

```
        df.insert(0, 'Protocol', protocol)

        # ── Model 1: Device Classification ──
        proba = classifier_pipeline.predict_proba(df)[0]
        pred_label = classifier_pipeline.predict(df)[0]
        conf = float(np.max(proba))
        device_str = DEVICE_LABELS.get(pred_label, f"Label-{pred_label}")

        print(f"[Model1] prediction = {device_str} ({pred_label}) (conf =
{conf:.2f})")

        if pred_label == 0 and conf >= MODEL1_CONF_THRESHOLD:
            # ── Model 2: Attack Detection ──
            atk_proba = attack_model.predict_proba(df)[0]
            atk_pred = attack_model.predict(df)[0]
            atk_conf = float(np.max(atk_proba))
            label_str = LABEL_MAP.get(atk_pred, f"Label-{atk_pred}")

            print(f"[Model2] attack = {label_str} ({atk_pred}) (conf =
{atk_conf:.2f})")

            packet = {
                "src_ip": src_ip,
                "src_port": int(src_port),
                "dst_ip": dst_ip,
                "dst_port": int(dst_port),
                "device_type": device_str,
                "attack_label": int(atk_pred),
                "attack_confidence": atk_conf
            }

            if atk_conf >= MODEL2_CONF_THRESHOLD:
                if atk_pred == MALICIOUS_LABEL:
                    streak['malicious'] += 1
                    print(f"[→] Malicious streak: {streak['malicious']}/5")
                    if streak['malicious'] >= 3:
                        try:
                            res = requests.post(MERN_ENDPOINT, json=packet,
timeout=2)
                            res.raise_for_status()
                            print(f"[⚠️] ALERT SENT: {packet}")
                        except RequestException as e:
                            print(f"[!] Alert send failed: {e}")
                        streak['malicious'] = 0
                else:
                    streak['malicious'] = 0
                    try:
                        res = requests.post(MERN_ENDPOINT, json=packet,
timeout=2)
```

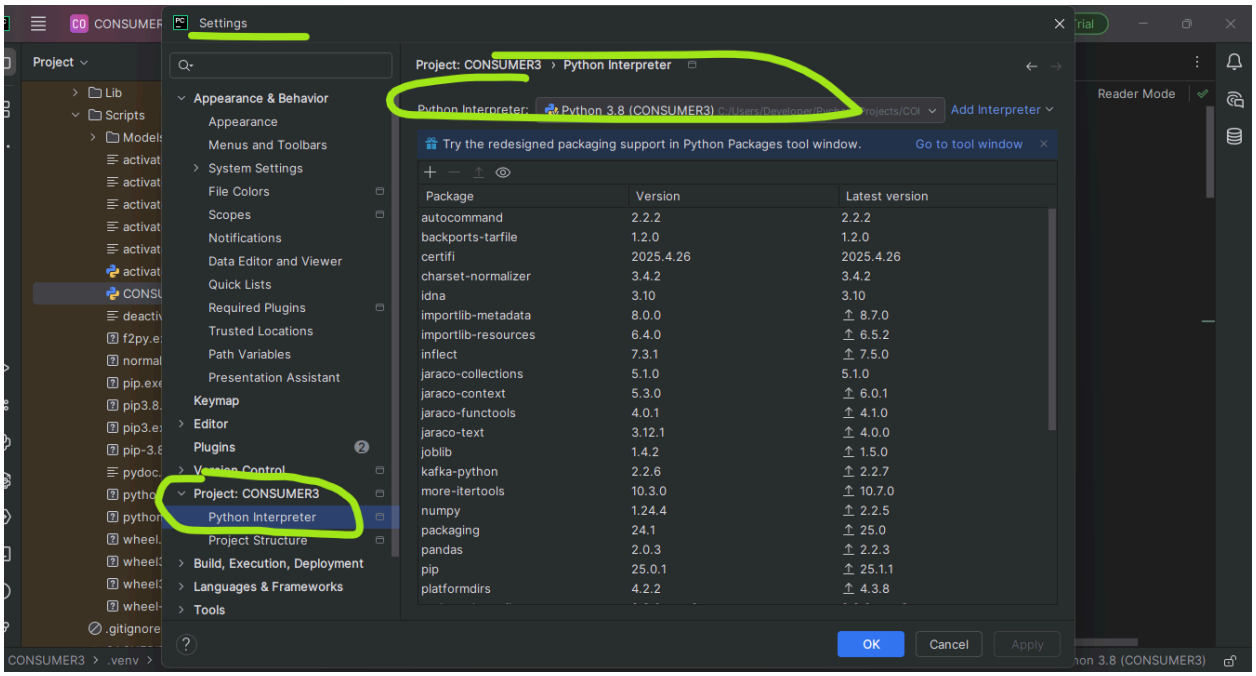
```
                    res.raise_for_status()
                    print(f"[✓] Normal packet sent: {packet}")
                except RequestException as e:
                    print(f"[!] Normal send failed: {e}")
            else:
                print(f"[→] Skipped MERN: Low confidence = {atk_conf:.2f}")
                streak['malicious'] = 0
        else:
            print(f"[→] Skipped attack model: device = {device_str}, conf =
{conf:.2f}")
            streak['malicious'] = 0

    except Exception as e:
        print(f"[✗] ERROR processing message: {e}")
```

Now open terminal in Pycharm and install these packages one by one as shown in the pic below.

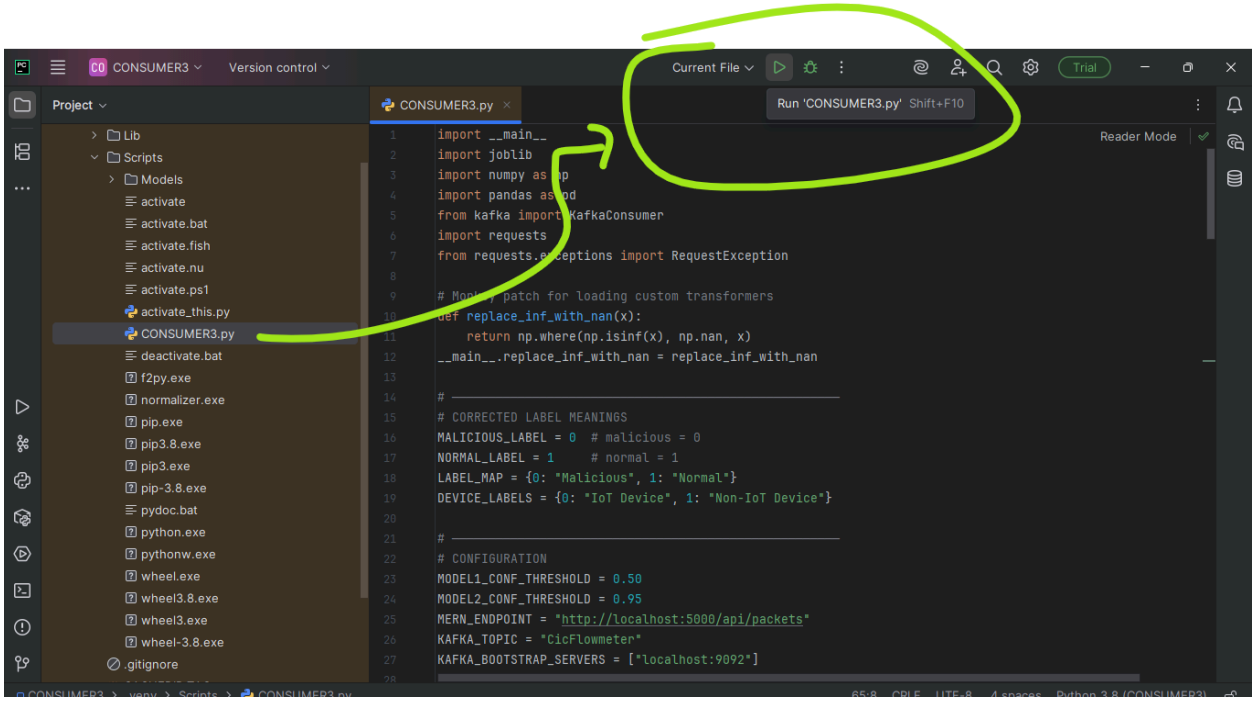
pip install joblib==1.4.2
pip install pandas==2.0.3
pip install kafka-python==2.2.6
pip install numpy==1.24.4
pip install requests==2.32.3
pip install xgboost==2.1.4
pip install scikit-learn==1.3.2



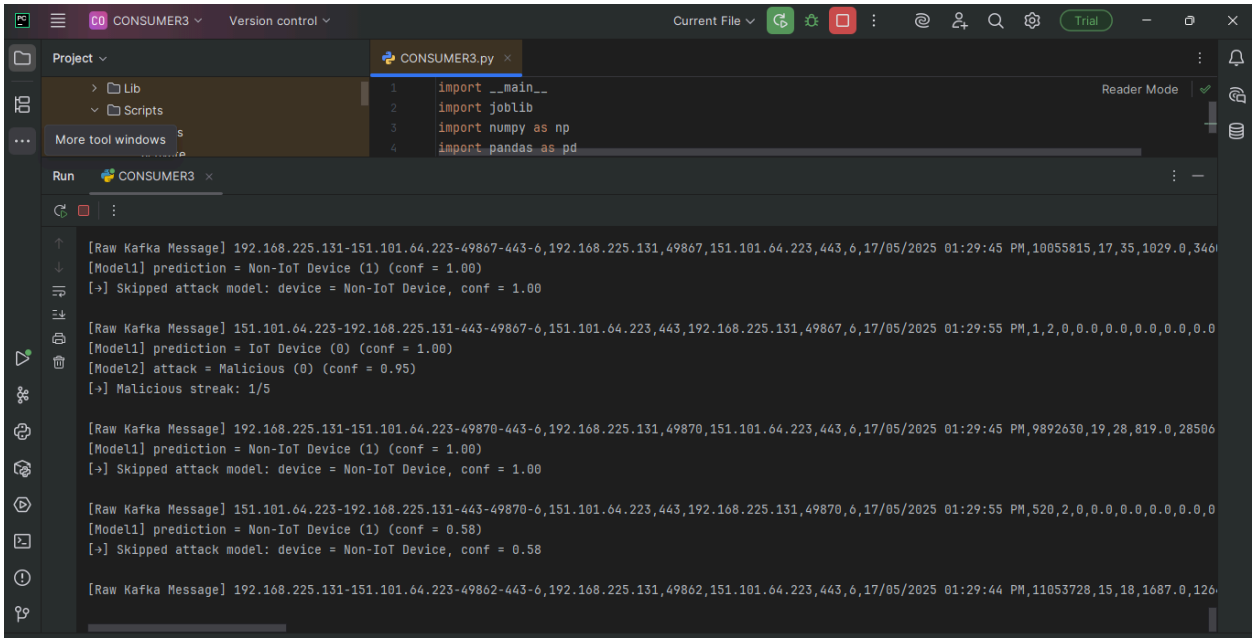
Now make sure your interpreter is set to python version 3.8.0



Now open CONSUMER3.py and run it.

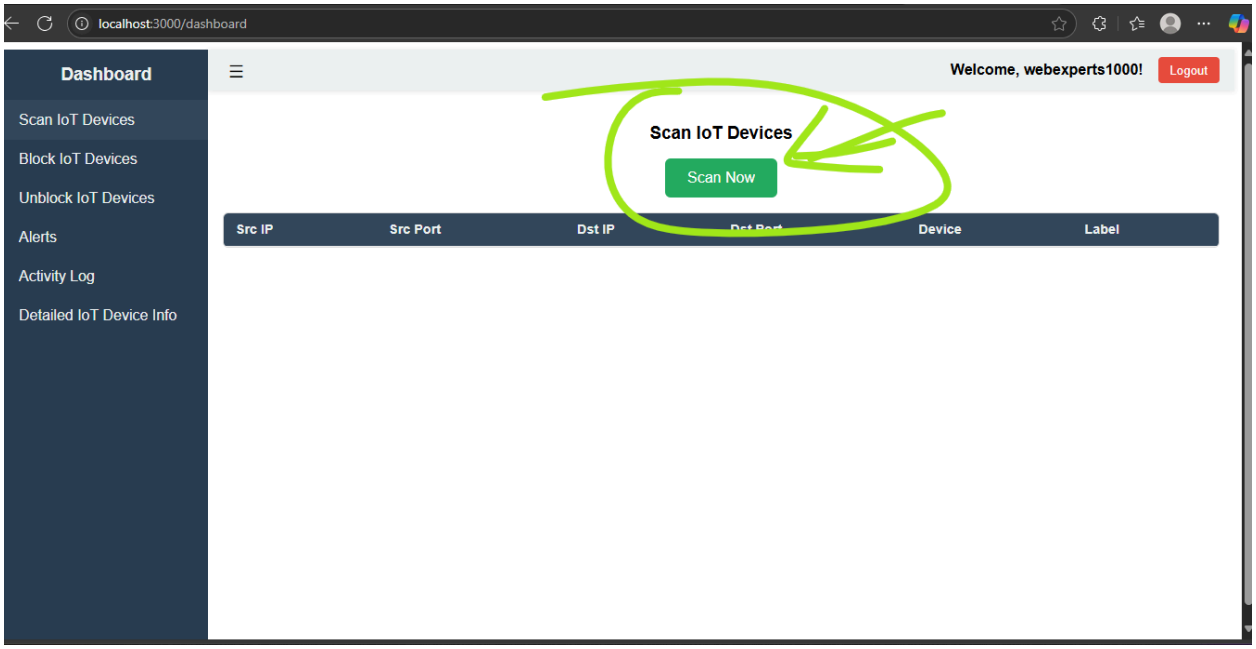


The packets should start showing in the console and the results of classification model and attack model

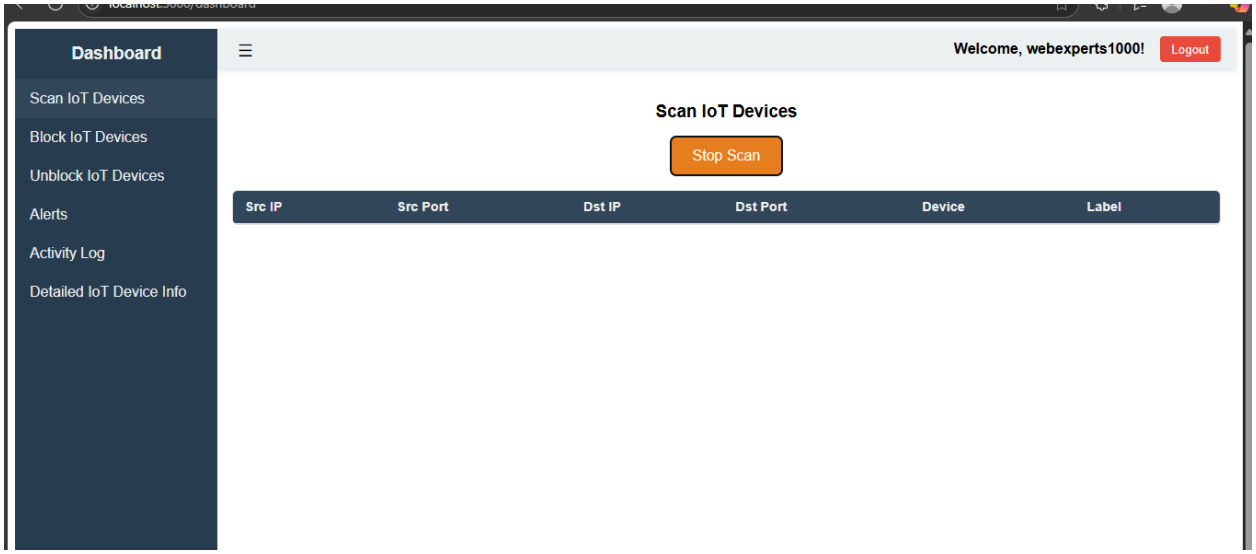


Since right now our laptop is only talking to the internet, the classification model hasn't detected any IoT Device yet.

Also, open the website, sign up and login, and click on Scan Now button.



And not much is showing up right now since none of the traffic is classified as IoT Device, as we haven't connected to our Raspberry Pi IoT device yet.
✅ Hence shown that our **classification model** is correctly mentioning that the current traffic is Non-IoT.

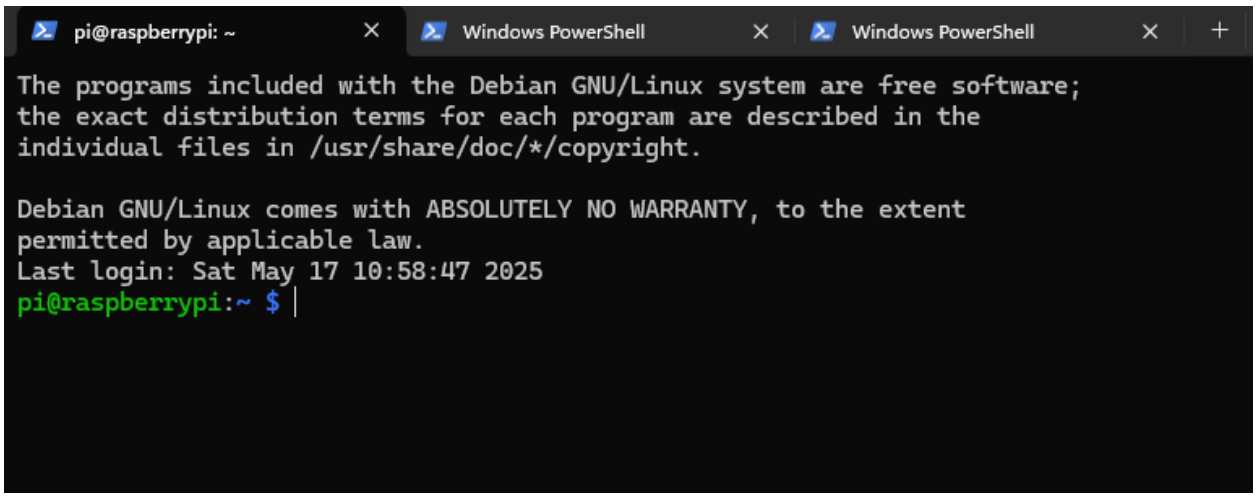


IoT Device Setup

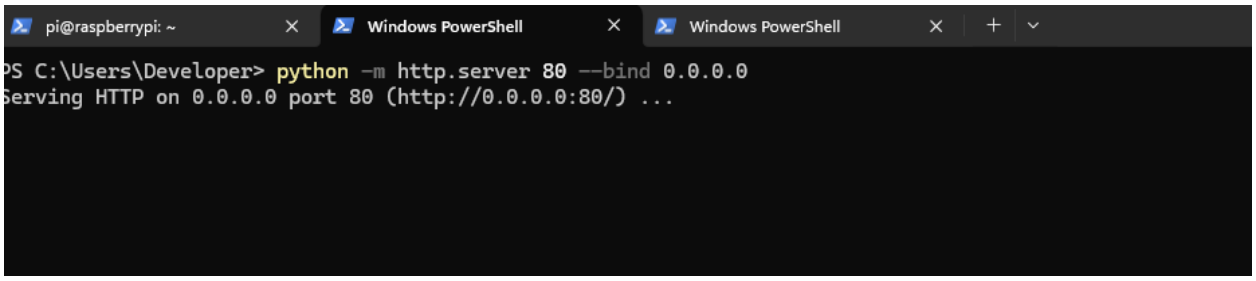
We will use a **Raspberry Pi** 3 Model B and turn it on and connect it to the same wifi our laptop is connected with, or otherwise if we aren't using wifi and using ethernet instead, then make sure to connect the pi and laptop both to the same ethernet network.

The Raspberry Pi is the IoT device.

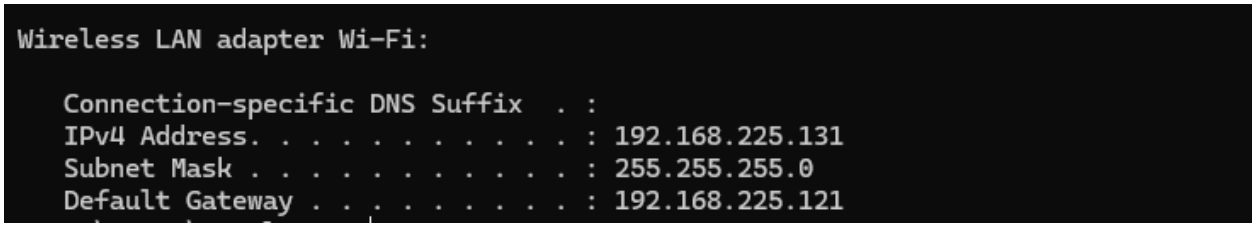
We will ssh into the pi instead of connecting the pi to a monitor and keyboard, as ssh is convenient.



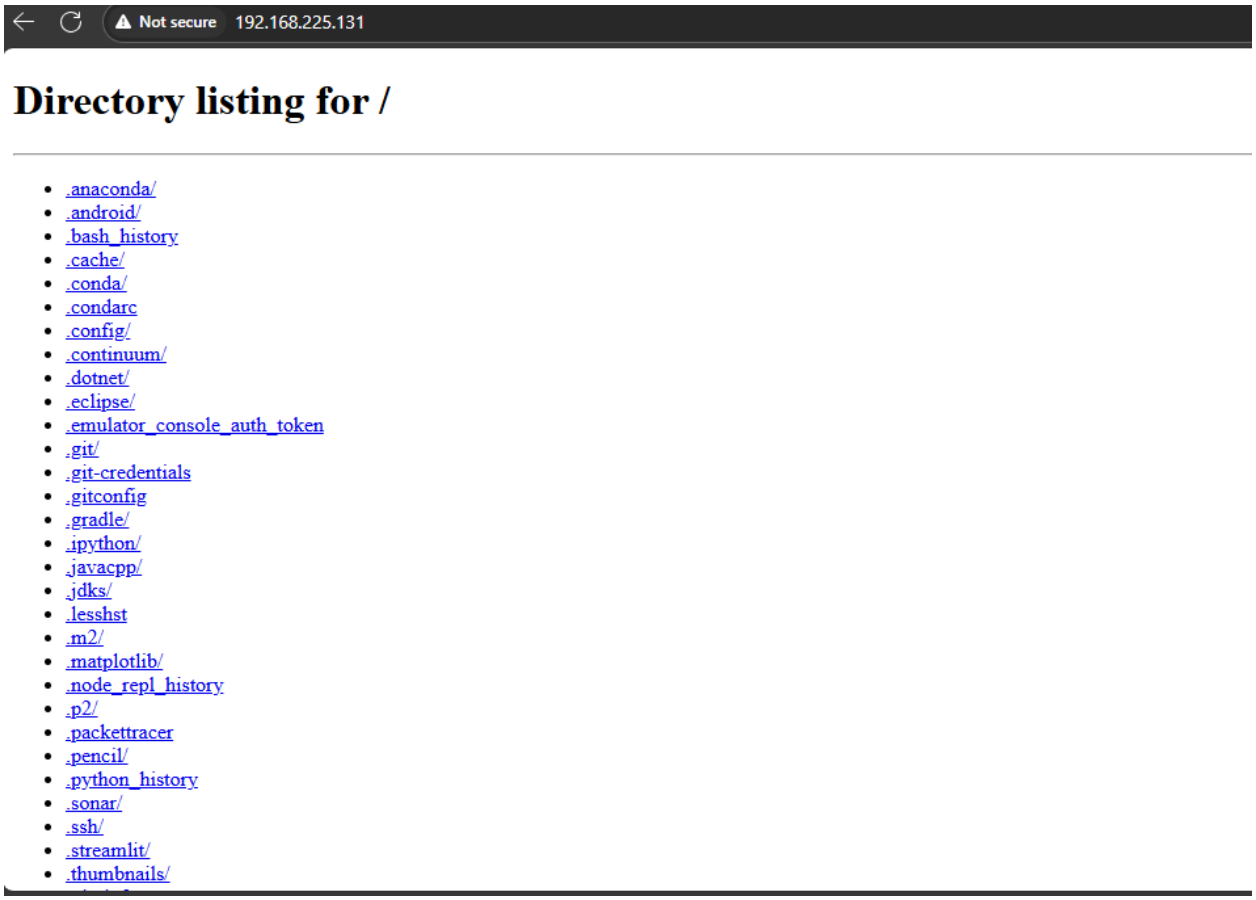
Now also open a new cmd tab or window, and turn on a http server using python, type **python -m http.server 80 --bind 0.0.0.0**



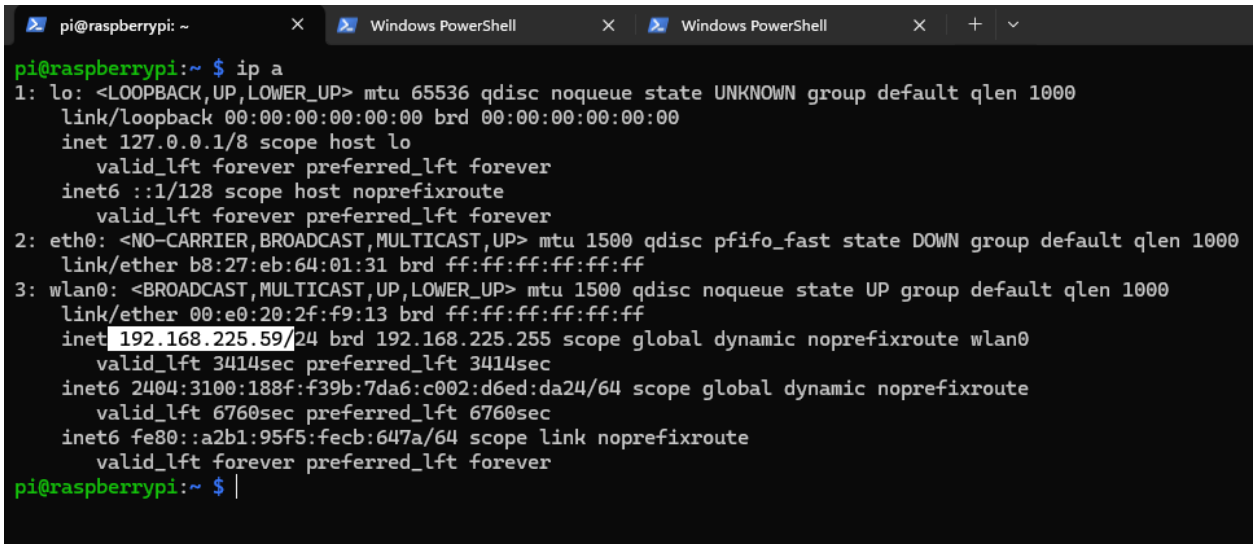
Now open a new tab in cmd, and type **ipconfig** and press enter. Check the local ipv4 of your interface.



In our case, our laptop's local ipv4 is 192.168.225.131
Let's open http://192.168.225.131 in a browser tab.



Now let's check our pi's local ipv4. Go to the ssh terminal or cmd, and type **ip a** and enter.

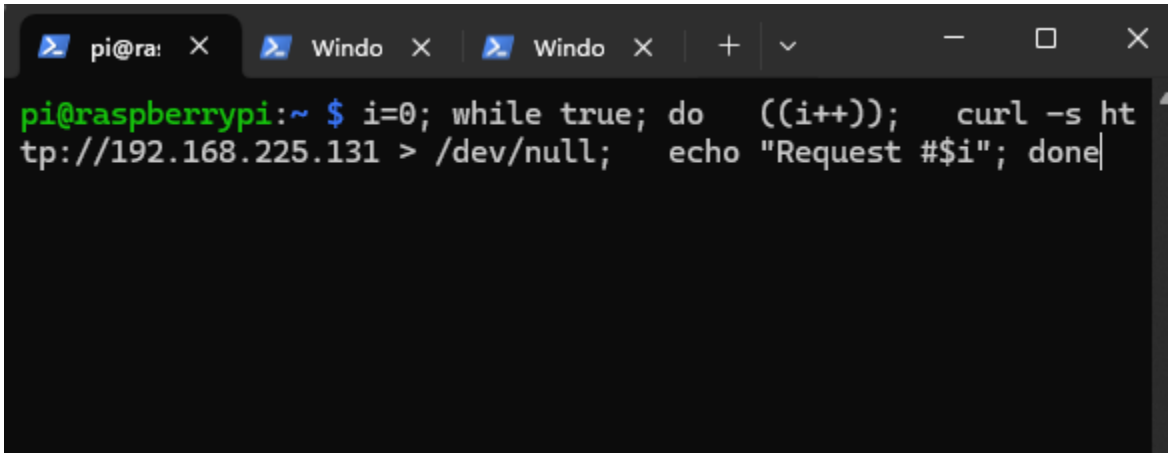


Our pi's local ipv4 is 192.168.225.59

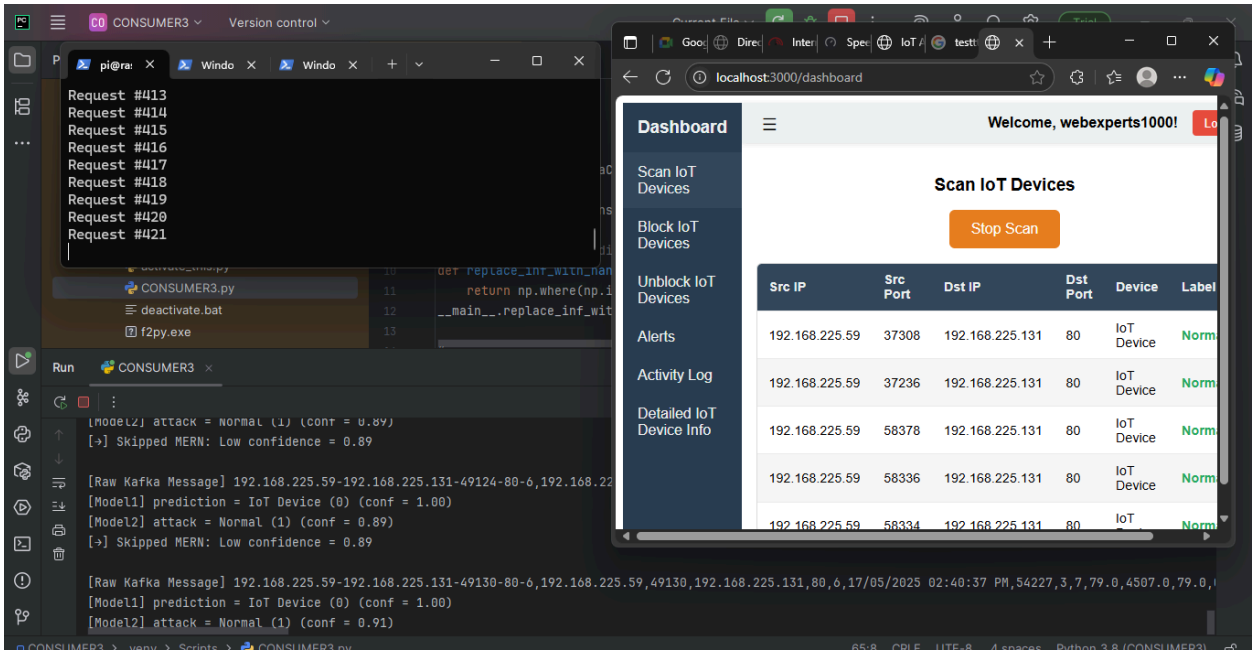
Now let's send packets from IoT Device Raspberry Pi to our laptop's http server.
We will use curl command, and send tons of **safe** packets, so go to the cmd tab in which you've connected to Pi via ssh, and type this command:

```
i=0; while true; do   ((i++));   curl -s http://192.168.225.131 > /dev/null;   echo "Request #$i"; done
```

Ofcourse make sure to change the ipv4 address in the above command to your own laptop's local ipv4.



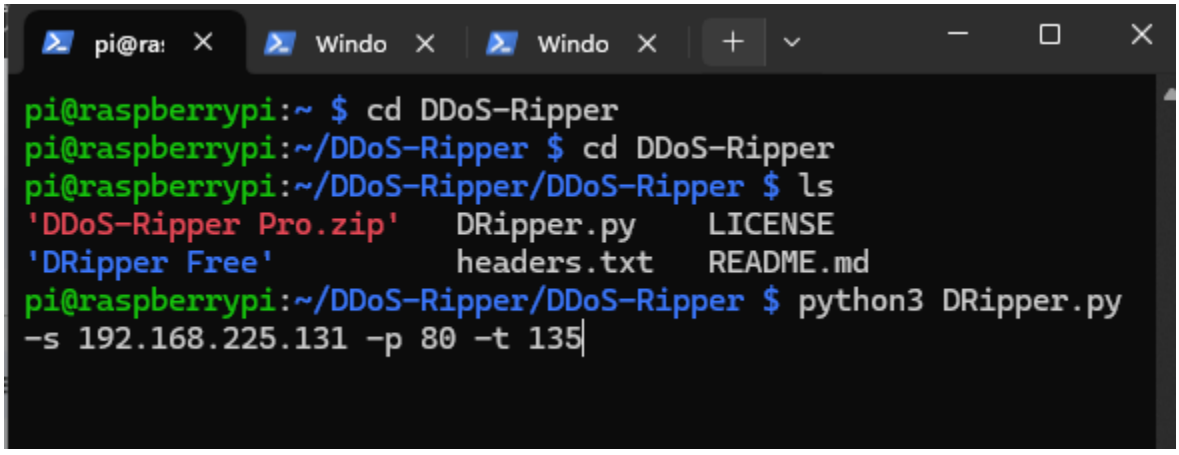
And press enter, so it will start sending tons of safe packets, and those safe packets would be received by CICFlowMeter which will send it to the Kafka server and Pycharm will show it on the terminal, and the website will update in real time.



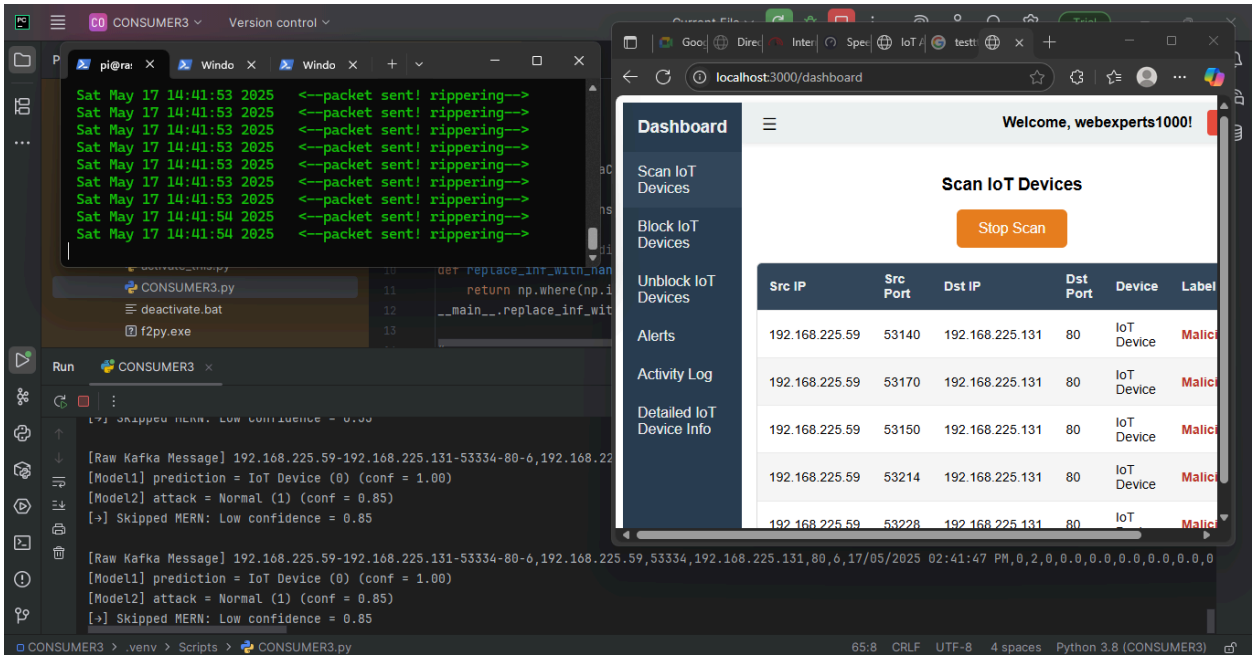
✅ Hence shown above that our both models, the **classification model**, and the **attack model**, are working correctly. It classifies IoT traffic correctly and only pushes that, and also the attack model correctly mentions that it's normal traffic.

Now download DDoS-Ripper to your IoT Device Raspberry Pi from https://github.com/palahsu/DDoS-Ripper

Extract the zip file downloaded, and navigate to DDoS-Ripper folder, and type this command to start the DDoS attack

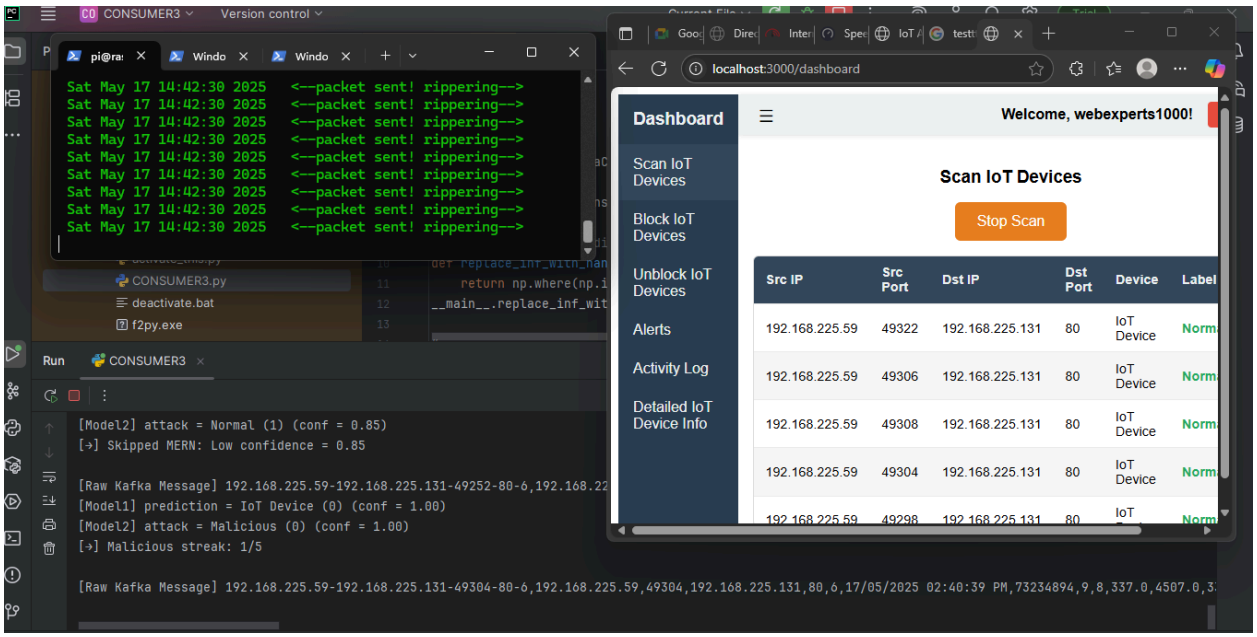


Once you press enter, the attack should start and the malicious packets should show up in the website.

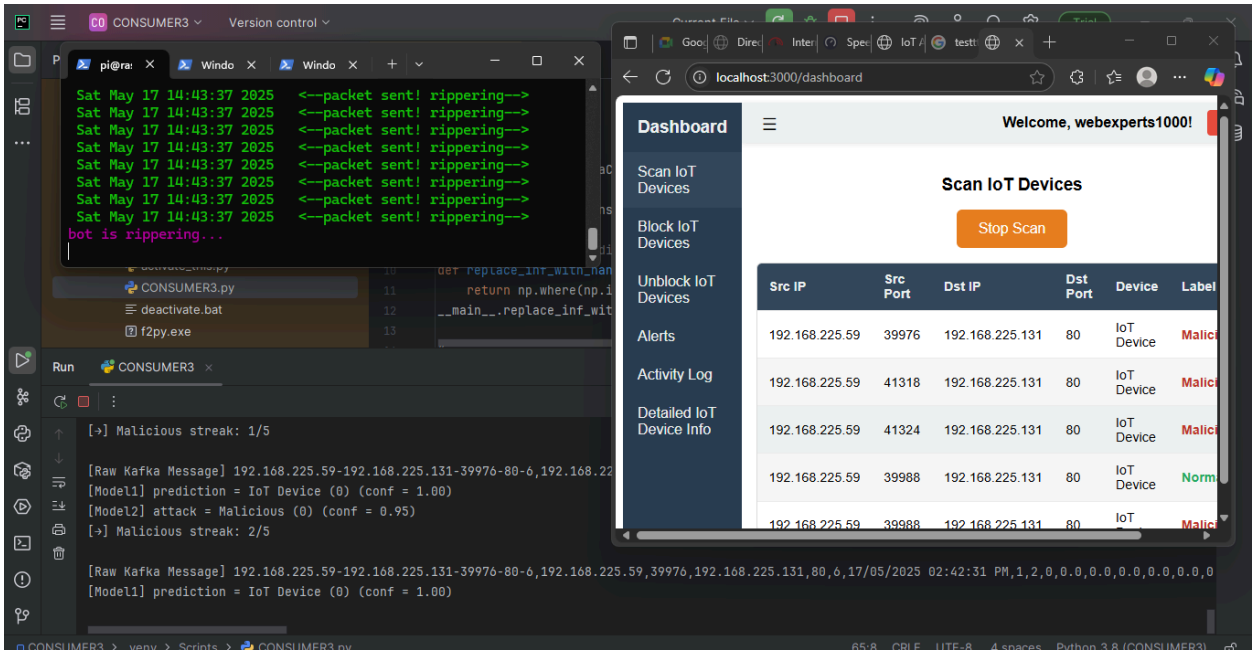


✅ Hence shown, that our **attack model** is correctly identifying it as malicious traffic.

After some time, the DDoS script stops sending malicious packets when it thinks that the server is down, and it basically sends some normal packets for some time.



And then suddenly the DDoS script begins rippering and sends malicious packets.



Conclusion

We have presented a pipeline for capturing, labeling, analyzing, and classifying network traffic for IoT security using CICFlowMeter and XGBoost. Beginning with raw packet capture through CICFlowMeter and controlled traffic generation, the workflow includes careful preprocessing, labeling based on IP and port behavior, and multiple stages of model training and evaluation. Visualizations were used to reveal behavioral patterns over time, and high-confidence classification was performed using a custom-trained XGBoost pipeline. The trained models were saved and reused for inference on unseen data, with reliable prediction filtering based on confidence thresholds. Overall, this approach provides an effective foundation for real-time attack detection and behavioral analysis of IoT networks.