Cyber Security Framework

Project Overview

Introduction

This project consists of three deep learning models aimed at detecting malicious activities across different domains. The models focus on:

1. **Malicious URL Detection** - Identifies phishing, XSS (Cross-Site Scripting), and SQL injection attacks in URLs.
2. **Malicious Network Traffic Detection** - Analyzes network traffic to detect malicious behaviors such as DDoS attacks or unauthorized access.
3. **Malicious Cloud Traffic Detection** - Monitors cloud network traffic to detect unusual and malicious activities in cloud environments.

Goal

The goal of this project is to develop a robust system capable of identifying and mitigating various cybersecurity threats in real-time using deep learning techniques.

Challenges
how to collect data which could be extracted for real-life scenario?

How to collect Data?

Public Data (URLs)

- **Source**: URLs are publicly available and can be gathered from various websites, datasets, or traffic data.
- **Model Use**: The model trained on URL data helps detect suspicious or malicious URLs based on features like length, domain type, and keywords, indicating phishing, malware, or other security risks.

Cloud Logs (CloudTrail)

- **Source**: Logs from cloud services (like AWS CloudTrail) provide detailed records of API calls, user activities, and resource usage within the cloud environment.
- **Model Use**: The model trained on CloudTrail logs focuses on identifying misconfigurations, unauthorized access, and unusual activity patterns in the cloud infrastructure, helping improve

security posture.

PCAP Files (Network Traffic)

- **Source**: PCAP files are network packet captures from sniffing tools like Wireshark. These files contain raw network traffic.
- **Model Use**: Models trained on network traffic (PCAP files) analyze patterns like unusual data flows, potential intrusions, and unauthorized network access, providing insights into real-time network security.

Each data type (URLs, cloud logs, and network traffic) serves a unique role in security detection, and by using them all, the model can provide a comprehensive view of potential threats across different layers of the system.

URL Data

URL data is widely available, and millions of URLs can be easily collected from various sources. However, the real challenge lies in effectively extracting meaningful features from these URLs for analysis or modeling purposes.

Feature Extraction Process for URL Analysis

Below is a Python function that takes a URL as input and returns a dictionary of the 22 features based on your requirements. These features can be useful for training models or analyzing URL patterns.

```
import pandas as pd
import re
from urllib.parse import urlparse
from tld import get_tld


def process_single_url(url):
    # 1. URL Length
    url_len = len(url)

    # 2. Process domain (TLD extraction)
    def process_tld(url):
        try:
            res = get_tld(url, as_object=True, fail_silently=False, fix_proto
col=True)
            return res.parsed_url.netloc
        except:
            return None
    domain = process_tld(url)

    # 3. Count the number of specific characters in URL
    features = ['@', '?', '-', '=', '.', '#', '%', '+', '$', '!', '*', '"',
',', '//']
    feature_counts = {feature: url.count(feature) for feature in features}

    # 4. Check for abnormal URL pattern (repeating hostname)
    def abnormal_url(url):
        hostname = urlparse(url).hostname
        return 1 if re.search(hostname, url) else 0

    abnormal_url_flag = abnormal_url(url)

    # 5. Check if the URL is using HTTPS
    def httpSecure(url):
        return 1 if urlparse(url).scheme == 'https' else 0

    https_flag = httpSecure(url)

    # 6. Count digits in the URL
    def digit_count(url):
        return sum(1 for char in url if char.isnumeric())

    digit_count_value = digit_count(url)

    # 7. Count letters in the URL
    def letter_count(url):
        return sum(1 for char in url if char.isalpha())

    letter_count_value = letter_count(url)

    # 8. Check if URL is from a shortening service
    def shortening_service(url):
        match = re.search(r'bit\.ly|goo\.gl|t\.co|tinyurl|adf\.ly|url4\.eu|sh
ort\.to|qr\.net|1url\.com', url)
        return 1 if match else 0

    shortening_flag = shortening_service(url)

    # 9. Count the number of directories in the URL path
    def no_of_dir(url):
        urldir = urlparse(url).path
        return urldir.count('/')
    dir_count = no_of_dir(url)

    # 10. Check for suspicious words in URL (e.g., 'login', 'paypal')
    def suspicious_words(url):
        match = re.search(r'PayPal|login|signin|bank|account|update|free|serv
ice|bonus|ebayisapi|webscr', url)
        return 1 if match else 0

    suspicious_flag = suspicious_words(url)

    # 11. Calculate hostname length
    hostname_length = len(urlparse(url).netloc)

    # 12. Count the number of uppercase letters in the URL
    upper_count = sum(1 for char in url if char.isupper())

    # 13. Count the number of lowercase letters in the URL
    lower_count = sum(1 for char in url if char.islower())

    # 14. Check if the URL has a "www" prefix
    has_www = 1 if 'www.' in url else 0

    # 15. Count number of subdomains (split by '.')
    subdomain_count = len(urlparse(url).hostname.split('.')) - 2 if urlparse
(url).hostname else 0

    # 16. Count the number of query parameters
    query_count = len(urlparse(url).query.split('&')) if urlparse(url).query
else 0

    # 17. Count the number of fragments in the URL
    fragment_count = 1 if urlparse(url).fragment else 0

    # 18. Check if the URL uses a port number
    has_port = 1 if urlparse(url).port else 0

    # 19. Count the number of slashes in the URL
    slash_count = url.count('/')

    # 20. Check if the URL uses a path
    has_path = 1 if urlparse(url).path else 0

    # 21. Check if the URL contains "http"
    contains_http = 1 if 'http' in url else 0
    # 22. Check if the URL contains a valid top-level domain
    valid_tld = 1 if process_tld(url) else 0

    # 23. Check if the URL contains a valid domain (e.g., example.com)
    has_valid_domain = 1 if domain else 0

    # 24. Check if the URL contains the string "secure"
    contains_secure = 1 if 'secure' in url else 0

    # 25. Check if the URL contains the string "login"
    contains_login = 1 if 'login' in url else 0

    # 26. Check if the URL contains the string "signup"
    contains_signup = 1 if 'signup' in url else 0

    # Combine all features into a dictionary
    features_dict = {
        'url_len': url_len,
        '@': feature_counts['@'],
        '?': feature_counts['?'],
        '-': feature_counts['-'],
        '=': feature_counts['='],
        '.': feature_counts['.'],
        '#': feature_counts['#'],
        '%': feature_counts['%'],
        '+': feature_counts['+'],
        '$': feature_counts['$'],
        '!': feature_counts['!'],
        '*': feature_counts['*'],
        ',': feature_counts[','],
        '//': feature_counts['//'],
        'abnormal_url': abnormal_url_flag,
        'https': https_flag,
        'digits': digit_count_value,
        'letters': letter_count_value,
        'Shortening_Service': shortening_flag,
        'count_dir': dir_count,
        'sus_url': suspicious_flag,
        'hostname_length': hostname_length
    }

    # Convert to a DataFrame (for easier handling and saving)
    df_single = pd.DataFrame([features_dict])

    #df_single['Category'] = -1  # Here, we set -1 because it's unknown for a
```

```
single URL
    # Save to CSV
    df_single.to_csv('single_url_test.csv', index=False)

    return df_single


# Example usage:
url = "http://example.com/login?user=test"
df_single = process_single_url(url)
print(df_single)
```

Explanation of Features

1. **url_len**: The total length of the URL. Longer URLs may sometimes indicate complex queries or potential obfuscation.
2. **@**: Count of the "@" symbol in the URL. Its presence might indicate the URL is an email address or the use of mail-related services.
3. **?**: Counts the "?" symbol, typically found in URLs with query parameters.
4. : Counts hyphens, which might separate parts of the domain name or path.
5. **=**: Counts the equal sign ("="), often used in URL parameters to assign values to variables.
6. **.**: Counts the number of periods (dots), which are common in domain names and file extensions.
7. **#**: Counts the hash (#) symbol, commonly used for fragment identifiers or anchors within a page.
8. **%**: Counts percent signs, which are used in URL encoding for special characters.
9. **+**: Counts plus signs, often used to represent spaces in URLs.
10. **$**: Counts the dollar sign, which may be associated with financial services or scams.
11. **!**: Exclamation mark count, which may be used to highlight or exaggerate a URL's purpose.
12. : Wildcard character count, although rare in URLs, it may be used in domain names or queries.
13. **,**: Comma count, which can be found in certain data URLs or query parameters.
14. **//**: Double-slash occurrences, which can identify the start of the domain or protocol part of the URL.
15. **abnormal URL**: A flag to check if the URL is an IP address (an abnormal pattern in most cases).
16. **https**: A binary feature indicating whether the URL uses the secure HTTPS protocol (1 for HTTPS, 0 for HTTP).
17. **digits**: Count of numeric digits in the URL, which might indicate the presence of a session ID or other identifiers.

18. **letters**: Count of alphabetic characters in the URL, which might be indicative of human-readable names or patterns.
19. **Shortening Service**: A binary feature indicating if the URL uses a known URL shortening service (1 for shortening, 0 for others).
20. **count Dir**: Counts the number of directory-like components in the URL path (indicated by slashes).
21. **sus_url**: A flag for suspicious keywords commonly used in phishing URLs (e.g., PayPal, login).
22. **hostname length**: The length of the domain name (hostname) in the URL, which may indicate its complexity or legitimacy.

Usage Example:

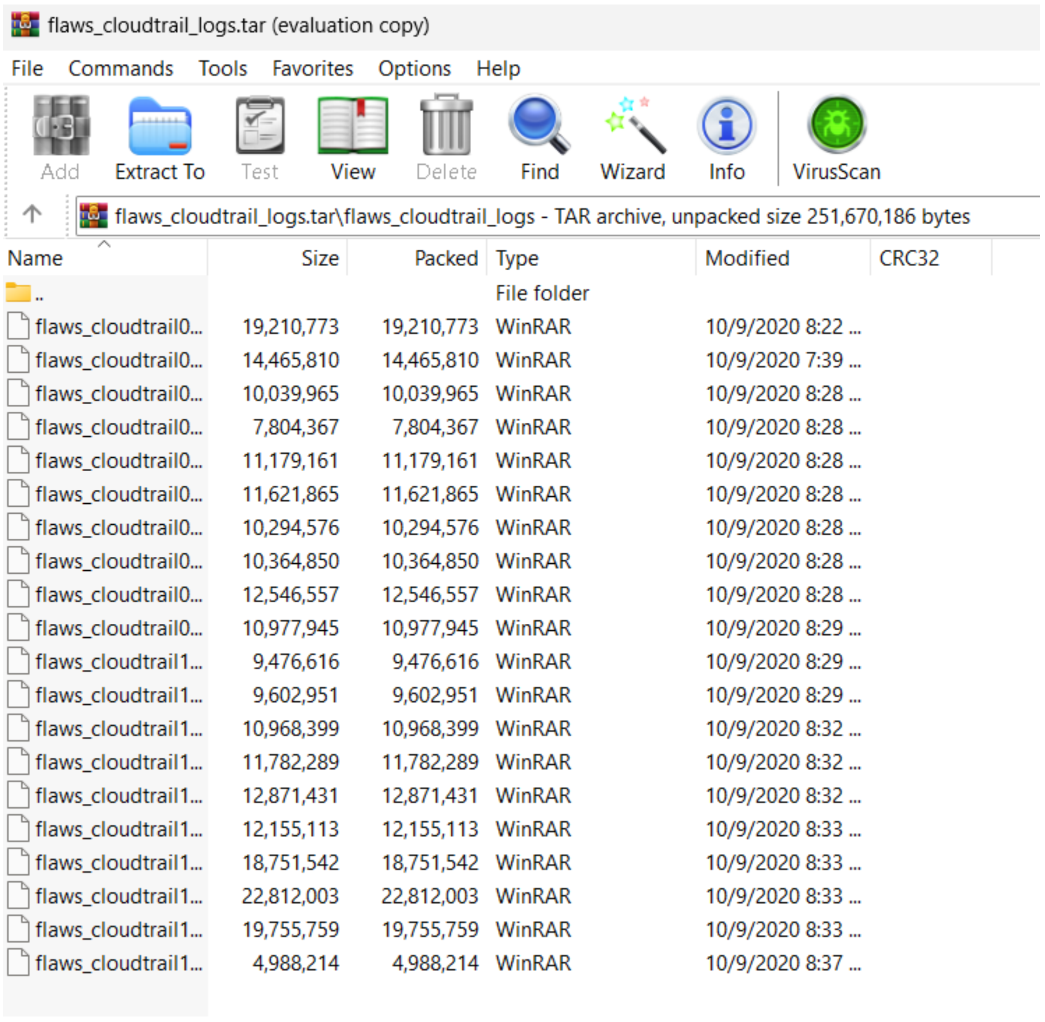
```
python
Copy code
url = "https://www.example.com/login?user=admin&pass=12345"
features = extract_url_features(url)
print(features)
```

This function returns a dictionary with the 22 features, which you can then use for further analysis, training a machine learning model, or other purposes.

Cloud Data

Collecting valid data for training proved to be a real challenge. After considerable research, we discovered that CloudTrail logs provided the most reliable and comprehensive dataset for our needs. In the cloud environment, security issues are primarily caused by misconfigurations rather than traditional network attacks. By training our model using CloudTrail logs, based on best practice configurations, we can effectively detect misconfigurations and enhance security.

AWS CloudTrail is a service that enables governance, compliance, and operational auditing by capturing and recording AWS API calls. It provides detailed logs of user activity, offering deep insights into resource usage and account actions across AWS services. CloudTrail is an essential tool for monitoring and securing AWS environments by logging all actions taken on cloud resources.


flaws_cloudtrail_logs.tar (evaluation copy)
File Commands Tools Favorites Options Help
Add Extract To Test View Delete Find Wizard Info VirusScan

flaws_cloudtrail_logs.tar\flaws_cloudtrail_logs - TAR archive, unpacked size 251,670,186 bytes

Name	Size	Packed	Type	Modified	CRC32
..			File folder		
flaws_cloudtrail0...	19,210,773	19,210,773	WinRAR	10/9/2020 8:22 ...	
flaws_cloudtrail0...	14,465,810	14,465,810	WinRAR	10/9/2020 7:39 ...	
flaws_cloudtrail0...	10,039,965	10,039,965	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	7,804,367	7,804,367	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	11,179,161	11,179,161	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	11,621,865	11,621,865	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,294,576	10,294,576	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,364,850	10,364,850	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	12,546,557	12,546,557	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,977,945	10,977,945	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	9,476,616	9,476,616	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	9,602,951	9,602,951	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	10,968,399	10,968,399	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	11,782,289	11,782,289	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	12,871,431	12,871,431	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	12,155,113	12,155,113	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	18,751,542	18,751,542	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	22,812,003	22,812,003	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	19,755,759	19,755,759	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	4,988,214	4,988,214	WinRAR	10/9/2020 8:37 ...	

after that we need to change this format for csv file to work in it “and we will do it in testing’

```
import pandas as pd
import json

with open(r'test.json', 'r') as file:
    data = json.load(file)

# If the JSON is a list of records and not a dictionary with a 'Records' key, nc
df = pd.json_normalize(data)

# Save the DataFrame to CSV
df.to_csv('output_json.csv', index=False)
```

```
# Show the first few rows of the DataFrame
print(df.head())
```

Network Data

In network security, we often work with real-world datasets, However, to test a machine learning model with this data, we need to convert **PCAP files into CSV format**. This conversion process involves **extracting relevant features** from the raw network traffic, such as IP addresses, ports, packet length, protocols, and timestamps. These features are then structured in a CSV format, which can be fed into deep learning model for testing and analysis.

and this’s code to convert from pcap to csv

```
import pyshark
import csv
import argparse
from collections import defaultdict


def extract_custom_fields(packet, flow_data):
    """Extract custom fields from a packet."""
    try:
        # Extract common fields
        frame_time_delta = float(packet.frame_info.time_delta)  # Time delta in
        flow_data['Flow Duration'] += frame_time_delta * 1e6  # Convert to micro

        # Extract packet direction (forward or backward)
        if hasattr(packet, 'ip') and hasattr(packet, 'tcp'):
            if 'src' in packet.ip.field_names and 'dst' in packet.ip.field_names
                if packet.ip.src < packet.ip.dst:
                    flow_data['Total Fwd Packets'] += 1
                    flow_data['Fwd Packets'] += int(packet.tcp.len or 0)
                else:
                    flow_data['Bwd Packets'] += int(packet.tcp.len or 0)

        # SYN/FIN flags
        if hasattr(packet, 'tcp'):
            flow_data['SYN Flag Count'] += int(packet.tcp.flags_syn == '1')
            flow_data['FIN Flag Count'] += int(packet.tcp.flags_fin == '1')
        # Update timestamps for Active Mean and Idle Mean
        timestamp = float(packet.frame_info.time_epoch)
        flow_data['timestamps'].append(timestamp)

    except AttributeError:
        pass


def finalize_features(flow_data):
    """Calculate derived features like Active Mean and Idle Mean."""
    timestamps = flow_data['timestamps']

    if len(timestamps) > 1:
        time_differences = [timestamps[i] - timestamps[i - 1] for i in range(1,

        # Active Mean: Time differences less than 1 second
        active_times = [t for t in time_differences if t < 1]
        flow_data['Active Mean'] = sum(active_times) / len(active_times) if acti

        # Idle Mean: Time differences greater than or equal to 1 second
        idle_times = [t for t in time_differences if t >= 1]
        flow_data['Idle Mean'] = sum(idle_times) / len(idle_times) if idle_times
    else:
        flow_data['Active Mean'] = 0
        flow_data['Idle Mean'] = 0

    # Remove timestamps (no need to output)
    del flow_data['timestamps']


def pcap_to_csv(input_file, output_file):
    """Convert PCAP file to CSV with custom fields."""
    # Open the PCAP file
    capture = pyshark.FileCapture(input_file)

    headers = [
        'Flow Duration', 'Total Fwd Packets', 'Fwd Packets', 'Bwd Packets',
        'Flow IAT', 'SYN Flag Count', 'FIN Flag Count', 'Active Mean', 'Idle Mea
    ]

    with open(output_file, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=headers)
        writer.writeheader()
        # Flow data storage
        flow_data = defaultdict(lambda: 0)
        flow_data['timestamps'] = []

        for packet_number, packet in enumerate(capture, start=1):
            extract_custom_fields(packet, flow_data)

            if packet_number % 100 == 0:
                print(f"Processed {packet_number} packets")

        # Finalize and write features
        finalize_features(flow_data)
        flow_data['Label'] = 'BENIGN'  # Update this dynamically if needed
        writer.writerow(flow_data)

    print(f"Conversion complete. Output saved to {output_file}")


input_file = "test.pcap"  # Replace with your PCAP file path
output_file = "output_pcap.csv"  # Replace with your desired output CSV path
pcap_to_csv(input_file, output_file)
```

The data collection process and answering how we will use and test these models is complete. Now, let's dive deeper into the models.

Documentation for URL Classification Model using Neural Networks

Overview

This script implements a classification model to detect different types of URLs (benign, defacement, phishing, malware). The data is processed from a CSV file containing URLs, and a neural network model is used to predict the type of URL based on various extracted features.

Data Preprocessing Steps

1. **Reading the Data:**
    - The data is loaded from the file `"url.csv"`, and its shape and structure are printed.
2. **Feature Engineering:**

- **URL Length:** A new column is added to represent the length of each URL.
- **TLD Processing:** The top-level domain (TLD) is extracted using `get_tld()`.
- **Feature Extraction:**
  - Several features are extracted from the URL such as the number of certain special characters (`@`, `?`, , etc.), whether the URL is using HTTPS, the number of digits and letters, whether the URL uses shortening services, and the number of directories in the path.
- **Abnormal URL:** A feature is added to flag URLs with abnormal hostname patterns.
- **Suspicious Words:** A feature is added to detect suspicious words in the URL related to phishing or scam attempts (e.g., "login", "paypal").
- **Hostname Length:** A feature is created for the length of the hostname.

3. **Target Column Transformation:**
   - The target column `type` is mapped to numeric categories for model training (`benign: 0, defacement: 1, phishing: 2, malware: 3`).

Data Split and Preparation

- **Feature Columns (`x`):** All columns excluding the original URL, type, and domain columns are used as features.
- **Target Column (`y`):** The target column is the transformed `Category` column.
- **Train-Test Split:**
  - The dataset is split into a training set (80%) and a test set (20%) using `train_test_split()`.
- **PyTorch Tensors:**
  - The data is converted into PyTorch tensors for model training. Features are converted to `float32`, while labels are converted to `long`.

Neural Network Model

An advanced feedforward neural network (with 4 layers) is used to classify URLs into one of four categories (benign, defacement, phishing, malware). The model includes:

- **Batch Normalization**: Applied to each hidden layer to stabilize training.
- **Dropout**: Regularization technique used to prevent overfitting (with a rate of 40%).
- **Activation Function**: ReLU (Rectified Linear Unit) is used for the hidden layers.
- **Output Layer**: The final output layer uses the `CrossEntropyLoss` function for multi-class classification.

Model Training

- The model is trained for 25 epochs with an early stopping condition after 10 epochs without improvement in validation loss.

- **Optimizer:** Adam optimizer is used with a learning rate of 0.001 and weight decay for regularization.
- **Learning Rate Scheduler:** A scheduler is applied to adjust the learning rate after every 30 epochs.

Model Evaluation

After training, the model is evaluated on the test set using the following metrics:

- **Accuracy**
- **Precision**
- **Recall**
- **F1 Score**

These metrics are printed for the test set, and a confusion matrix is generated to visualize the classification performance.

Visualization

- **Distributions:** The distribution of features such as URL length and the count of suspicious characters are visualized using seaborn.
- **Confusion Matrix:** A heatmap of the confusion matrix is plotted to assess the model's performance on the test set.

Code Walkthrough

Data Preprocessing

```
python
Copy code
df['url_len'] = [len(url) for url in df.url]  # Add URL length feature
df['domain'] = df['url'].apply(lambda i: process_tld(i))  # Extract TLD (Top-
Level Domain)
df['https'] = df['url'].apply(lambda i: httpSecure(i))  # Check if URL uses H
TTPS
df['digits'] = df['url'].apply(lambda i: digit_count(i))  # Count digits in t
he URL
df['letters'] = df['url'].apply(lambda i: letter_count(i))  # Count letters i
n the URL
df['Shortening_Service'] = df['url'].apply(lambda x: Shortening_Service(x))
# Check for shortening service
df['count_dir'] = df['url'].apply(lambda i: no_of_dir(i))  # Count directorie
s in the URL path
df['sus_url'] = df['url'].apply(lambda i: suspicious_words(i))  # Check for s
```

```
uspicious words
```

Model Architecture

```
python
Copy code
class AdvancedNNModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(AdvancedNNModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)  # Output layer without activation
        return x
```

Training Loop

```
python
Copy code
for epoch in range(epochs):
    # Train model
    model.train()
    # Validation
    model.eval()
```

Model Evaluation

```
python
Copy code
# Final Test Metrics
accuracy = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
precision = precision_score(y_test_tensor.numpy(), y_pred.numpy(), average='w
eighted')
recall = recall_score(y_test_tensor.numpy(), y_pred.numpy(), average='weighte
d')
f1 = f1_score(y_test_tensor.numpy(), y_pred.numpy(), average='weighted')
```

Confusion Matrix

```
python
Copy code
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Class 0', 'C
lass 1', 'Class 2', 'Class 3'], yticklabels=['Class 0', 'Class 1', 'Class 2',
'Class 3'])
```

Conclusion

This script demonstrates how to process URL data, extract meaningful features, and apply a neural network model for classification tasks. The resulting model can classify URLs into different categories, which is useful for detecting phishing, malware, and other malicious web activities.

Documentation for the LSTM-based Network Intrusion Detection System (IDS) Code

This code implements a network intrusion detection system (IDS) using an LSTM (Long Short-Term Memory) model, designed to classify network traffic based on various attack types. The model is trained and tested on a dataset containing various types of network attacks and benign traffic. The code utilizes PyTorch for deep learning, alongside libraries like Pandas, Scikit-learn, and Seaborn for data preprocessing, visualization, and model evaluation.

Key Sections of the Code:

1. **Data Loading and Exploration**
   - Loads the dataset from a CSV file (`network_ids_data.csv`) using `pandas.read_csv()`.
   - Inspects the data using `.head()`, `.info()`, and `.shape()` to get an overview of the structure and dimensions of the data.
   - Displays missing values using `sns.heatmap()` to visualize null values in the dataset.
   - Removes duplicate records with `drop_duplicates()` and checks for infinite values.
2. **Data Preprocessing**
   - **Handling Missing or Infinite Values**: Identifies any infinite values in the numeric columns and reports them.

- **Feature Scaling**: Uses `MinMaxScaler` from Scikit-learn to scale the numeric features between 0 and 1.
- **Label Encoding**: The target class labels (`Label`) are encoded using `LabelEncoder` from Scikit-learn to convert categorical labels into numerical values.

3. **Train-Validation-Test Split**
   - The data is split into training, validation, and testing datasets using `train_test_split` from Scikit-learn.
   - The datasets are converted into PyTorch tensors and moved to the device (GPU if available).
4. **PyTorch DataLoader**
   - Converts the data into `TensorDataset` and creates `DataLoader` instances for the training, validation, and test sets. The `DataLoader` is used to load data in batches for efficient training.
5. **LSTM Model Definition**
   - **LSTM Model Architecture**: Defines a custom LSTM model class (`LSTMModel`) using `nn.Module` from PyTorch. The model consists of:
     - An LSTM layer for sequential learning.
     - A fully connected (`fc`) layer for classification.
   - **Forward Pass**: The `forward()` function ensures that the input data has the correct dimensions (3D: batch_size x sequence_length x input_size). It processes the input through the LSTM and outputs the predictions.
6. **Model Training**
   - **Loss Function and Optimizer**: The model uses `CrossEntropyLoss` as the loss function for multi-class classification and `Adam` optimizer for gradient descent.
   - **Training Loop**: Trains the model for 10 epochs. After each epoch, the model is evaluated on the validation set. Early stopping is implemented to prevent overfitting, with patience set to 5 epochs.
   - **Monitoring Performance**: The training loss and validation accuracy are printed at each epoch to monitor progress.
7. **Testing and Evaluation**
   - After training, the model is evaluated on the test set. The test accuracy is computed and a detailed classification report is printed using Scikit-learn's `classification_report()` function.
   - The trained model is saved to a file (`trai_model.pth`) using `torch.save()`.
8. **Model Inference on New Data**
   - The model is loaded from the saved file (`trai_model.pth`) using `load_state_dict()`.
   - New data is read from a test CSV file (`network_test.csv`) and preprocessed similarly (e.g., reshaping to fit the LSTM input format).

- The model is used for inference on the new data, and the predicted class probabilities are obtained using the softmax function.
- The predicted class index and the corresponding class probabilities are printed.

9. **Label Mapping**
   - A dictionary (`label_map`) is created to map class labels to their encoded numerical values.
   - The code includes an example of decoding an encoded label back to its corresponding class name.

Example Output:

- **Training Process**: For each epoch, the training loss, validation loss, and validation accuracy are printed.

```
yaml
Copy code
Epoch 1/10, Train Loss: 0.5231, Val Loss: 0.4324, Val Accuracy: 0.8134
...
```

- **Test Accuracy**:

```
mathematica
Copy code
Test Accuracy: 0.8232
```

- **Predictions for New Data**:

```
kotlin
Copy code
Predicted class index: 4
Predicted probabilities: tensor([0.0982, 0.0823, 0.0904, 0.0873, 0.1071,
0.0892, 0.0811, 0.0809, 0.0874,
   0.1075, 0.0862, 0.0858, 0.0876, 0.0871])
```

- **Class Label Mapping**:

```
yaml
Copy code
Mapping between class names and encoded labels:
BENIGN: 0
DoS Hulk: 1
```

```
FTP-Patator: 2
PortScan: 3
DDoS: 4
...
```

Dependencies:

- `numpy`
- `pandas`
- `seaborn`
- `matplotlib`
- `sklearn`
- `keras`
- `torch`
- `tqdm`
- `plotly`

Notes:

- **LSTM Input Shape**: The model expects input data to be in the form of (batch_size, sequence_length, input_size). If the input data is 2D (samples x features), it is reshaped to have a sequence length of 1 (`samples x 1 x features`).
- **Early Stopping**: Stops the training if the validation loss does not improve for 5 consecutive epochs.
- **Softmax**: Softmax is applied to the model's predictions during inference to get class probabilities.

To Run:

1. Ensure all dependencies are installed (`pip install torch scikit-learn pandas seaborn matplotlib tqdm plotly`).
2. Place the dataset (`network_ids_data.csv`, `network_test.csv`) in the same directory.
3. Run the code, monitor training, and use the model for prediction.

This approach leverages LSTM's ability to model sequential data and is suited for time-series problems such as network traffic analysis.

Documentation for Cloud Intrusion Detection System (IDS) Code

Overview

This script preprocesses data and trains a machine learning model using PyTorch to perform binary classification. It combines numerical and categorical data for prediction. The primary task involves embedding categorical features, standardizing numerical features, and training a neural network model.

Key Dependencies

- **NumPy**: For numerical operations.
- **Pandas**: For data manipulation and preprocessing.
- **Seaborn**: For visualizing data.
- **Scikit-learn**: For data preprocessing and splitting.
- **PyTorch**: For building and training the neural network.

Code Breakdown

1. Data Loading and Inspection

- **`pd.read_csv`**: Loads the dataset (`sampled_data.csv`) into a Pandas DataFrame.
- **Shape Inspection**:
  - `data.shape`: Outputs the dimensions of the dataset.
  - `data.info()`: Summarizes the dataset, showing column names, data types, and non-null counts.

---

2. Preprocessing

- **Whitespace Removal**:
  - Strips leading/trailing whitespaces from column names using a dictionary comprehension and `data.rename()`.
- **Handling Missing Values**:
  - `data.dropna()`: Removes rows with missing values.
  - `data.isna().sum()`: Counts missing values in each column.
- **Feature Engineering**:
  - Converts `requestParametersinstanceType` into binary labels (`0` for "NotApplicable", `1` for breached).

---

3. Data Preparation

- **Feature Splitting**:
  - **Numerical Features**: Only `eventVersion`.
  - **Categorical Features**: Includes several columns, such as `userAgent`, `eventName`, `awsRegion`, etc.
- **Preprocessing**:

    - **Numerical Features**: Standardized using `StandardScaler`.
    - **Categorical Features**: Encoded using `LabelEncoder`.
- **Train-Test Split**:
    - Data split into training (80%) and validation (20%) sets using `train_test_split`.

4. Dataset Class for PyTorch

- **Custom Dataset**:
    - The `MixedDataDataset` class extends `torch.utils.data.Dataset`.
    - Accepts numerical data, categorical data, and labels.
    - Implements `__len__` and `__getitem__` methods to index data.
- **DataLoader**:
    - Converts the dataset into batches for efficient training and validation.

5. Model Architecture

- `MyModel`:
    - A custom PyTorch model combining:
        - **Numerical Features**: Processed through fully connected layers with ReLU activation, batch normalization, and dropout.
        - **Categorical Features**: Embedded into fixed-size vectors using `nn.Embedding`.
    - Combines numerical and categorical features before passing through fully connected layers for binary classification.

6. Training and Evaluation

- **Training Loop**:
    - Processes training batches with `train_loader`.
    - Calculates binary cross-entropy loss (`nn.BCEWithLogitsLoss`).
    - Optimizes model parameters using `torch.optim.Adam`.
- **Evaluation Loop**:
    - Evaluates model performance on validation data.
    - Reports loss and accuracy.
- **Training Functionality**:
    - Uses `tqdm` for displaying progress bars.
    - Applies sigmoid activation to convert logits to probabilities for binary classification.

7. Execution

- Model is instantiated and moved to the GPU (if available).
- Model is trained over multiple epochs, with evaluation after each epoch.

How to Use

1. Ensure all required libraries are installed (`pip install numpy pandas seaborn scikit-learn torch tqdm`).
2. Replace `'sampled_data.csv'` with the path to your dataset.
3. Customize numerical and categorical columns based on the dataset structure.
4. Run the script to preprocess data, train the model, and evaluate its performance.