My Journey in Building a Network Anomaly Detection System

1. Introduction and Project Objective

As part of my endeavor to delve into network security and machine learning, I undertook the development of a Network Anomaly Detection System. My primary objective was to build a robust system capable of real-time monitoring of network traffic to identify unusual patterns, which could signal various security threats like data exfiltration, Distributed Denial-of-Service (DDoS) attacks, or malware infections. I aimed to leverage machine learning techniques to provide early warnings before potential malicious activities could escalate and cause significant damage.

2. Phase 1: Setting Up My Development Environment

My first step involved meticulously preparing my development environment, ensuring all necessary tools and libraries were correctly configured.

Leveraging my existing development setup, I utilized **Visual Studio Code (VS Code)** as my primary Integrated Development Environment (IDE). To enhance my Python development workflow within VS Code, I installed essential extensions, including the **Python extension** for core language support, the **Jupyter extension** for interactive notebook development, and **Pylance** for intelligent code completion and static analysis.

For dependency management, I created a **Python virtual environment** (`venv`). This crucial step isolated my project's libraries, preventing conflicts with other Python projects on my system. I ensured proper activation of this environment before proceeding with library installations.

Next, I installed the core Python libraries. My selections included **Pandas** and **NumPy** for data manipulation and numerical operations, **Matplotlib** and **Seaborn** for data visualization, and **Scikit-learn** for machine learning algorithms. Crucially, **Scapy** was installed for powerful packet manipulation and network traffic analysis. Early on, I encountered challenges with `tensorflow`'s incompatibility with Python 3.13, which I resolved by either adjusting my

Python version or simply proceeding with Scikit-learn's Isolation Forest as planned, as `tensorflow` wasn't strictly necessary for my chosen model. I also noted issues with `pcapy` and `dpkt` installation due to Python 3.13 compatibility, further solidifying my reliance on `scapy`.

Finally, for low-level packet capture, I installed **Wireshark**, an invaluable tool for network protocol analysis. Alongside it, **Npcap** was installed on my Windows system, which acts as the packet capture driver, ensuring my Python scripts could access live network interfaces. I confirmed its active status by checking its service.

3. Phase 2: Acquiring and Preparing Network Data

This phase was critical for transforming raw network data into a structured format suitable for machine learning.

I started by acquiring a **pre-recorded PCAP file** (dns.cap) from a public repository (Wireshark Sample Captures) for initial development. This allowed me to focus on data processing without the complexities of live capture initially.

My first custom script, pcap_parser.py, was developed to perform **data acquisition**. Its purpose was to read the raw PCAP file and extract **packet-level features** such as timestamp, source and destination IP addresses, protocol, ports (if applicable), and packet length. I used scapy to efficiently parse the packets and pandas to store the extracted data in a DataFrame, which was then saved as extracted_features.csv.

Following this, I moved to a Jupyter Notebook, preprocessing.ipynb, to perform **feature engineering and data preprocessing**. Here, I aimed to transform the raw features into more meaningful representations. My steps included:

- **Initial Cleaning:** Handling missing port values (filling with -1) and converting timestamps to datetime objects for easier manipulation.
- **Feature Engineering:** I engineered new features crucial for anomaly detection. This involved creating **time-based features** (hour of day, day of week, and minute of hour from timestamps) and **port presence indicators** (binary flags for common ports like HTTP, HTTPS, SSH). Most importantly, I developed **flow-based features** by grouping packets into bidirectional flows (based on sorted IP/port/protocol tuples) and calculating statistics like bidir_flow_duration, bidir_total_packets, and bidir_total_bytes. This was essential as many network anomalies manifest at the flow level rather than in single packets.

- **Preprocessing (Scaling and Encoding):** I rigorously prepared the data for machine learning. I used **Scikit-learn's ColumnTransformer** to apply different transformations to different feature types. Numerical features (like packet_length, flow statistics) were scaled using StandardScaler, while categorical features (like protocol_name, src_port, dst_port) were converted using OneHotEncoder. A significant challenge I encountered here was ensuring consistency, specifically with the minute_of_hour column being present during ColumnTransformer's fitting but sometimes inadvertently dropped later. I resolved this by meticulously refining the features_to_drop_for_preprocessing list to ensure only temporary columns were removed, and crucial features were consistently passed to the preprocessor.
- **Saving Outputs:** After successful preprocessing, I saved the transformed data as preprocessed_data.csv and, vitally, saved the **fitted ColumnTransformer object** itself as fitted_preprocessor.joblib. This ensures that any new data processed in later phases would be transformed using the exact same scaling parameters and encoding rules learned from the training data, maintaining pipeline consistency.

4. Phase 3: Developing My Anomaly Detection Model

With the preprocessed data ready, I proceeded to build and train my anomaly detection model.

My choice for the core algorithm was **Isolation Forest**, an unsupervised machine learning algorithm from Scikit-learn. I selected it because it is highly effective at identifying anomalies in high-dimensional datasets by isolating outliers rather than profiling normal data. As an unsupervised method, it was ideal for detecting potentially unknown security threats in unlabeled network traffic.

I developed the model_trainer.py script for this phase. This script loaded my preprocessed_data.csv and then initialized and trained the IsolationForest model. A key hyperparameter I configured was contamination, which represents the estimated proportion of anomalies in the training dataset (I initially set it to 0.01). The model then calculated an anomaly_score for each data instance.

Following model training, I focused on **thresholding**. For unsupervised models, an anomaly score needs a threshold to classify data as "normal" or "anomalous." I analyzed the distribution of anomaly scores (e.g., using histograms and box plots) and chose a percentile-based threshold (e.g., the 5th percentile) for initial classification. Any data point with an anomaly score at or below this threshold was flagged.

Finally, I saved my trained **IsolationForest model** as isolation_forest_model.joblib and the determined **anomaly threshold** as anomaly_threshold.txt. These saved assets are crucial for deploying the model in real-time.

5. Phase 4: Evaluating My Model's Performance

This phase was dedicated to assessing how well my trained anomaly detection model performed.

I created an evaluation.ipynb Jupyter Notebook for this purpose. In this notebook, I loaded my saved IsolationForest model, the anomaly_threshold, and the fitted_preprocessor.

For evaluation, I chose to process the dns.cap file again (though ideally, a separate, unseen dataset with known anomalies is preferred for realistic evaluation). My evaluation notebook replicated the pcap_parser.py logic to parse the raw PCAP, then applied the same feature engineering steps, and most importantly, used the *loaded fitted_preprocessor* to transform the data consistently. I paid close attention to resolving errors related to missing columns (like minute_of_hour) during this transformation by ensuring column lists matched exactly what the preprocessor was originally fitted on.

After preprocessing, I ran inference on this data to get anomaly scores and classifications. My evaluation primarily involved:

- **Qualitative Analysis:** I inspected the characteristics of the detected anomalies (preprocessed features, original context like IPs and ports) to understand why they were flagged.
- **Visualization:** I plotted the distribution of anomaly scores for the evaluation data, along with the chosen threshold, to visually assess how well anomalies stood out.

While full **quantitative evaluation** (using metrics like Precision, Recall, F1-score, ROC/AUC) requires a labeled test dataset (which was beyond the scope of this particular project's data acquisition), the qualitative analysis and visual inspection provided valuable insights into the model's behavior and sensitivity.

6. Phase 5: Implementing Real-time Monitoring and Alerting

This was the culmination of my project, where I applied the trained model to live network traffic.

I developed the realtime_monitor.py script for this phase. The primary objective was to continuously capture packets from a network interface, process them, and alert on detected anomalies.

A key challenge here was adapting the **flow-based model** (trained on aggregated flow statistics) to **per-packet real-time processing**. For simplicity in this project, I made a compromise: realtime_monitor.py captures and processes individual packets, extracting only packet-level features (timestamps, IPs, ports, length) and easily derivable time-based/port-presence features. Flow-based features (like bidir_flow_duration) were filled with default values (e.g., 0) as they require complex, stateful tracking of network conversations, which was beyond the scope of this tutorial. I acknowledged that a production system would require a more sophisticated, stateful flow aggregator.

The script loads the saved model, threshold, and fitted_preprocessor. It then uses scapy.sniff to capture packets from a specified network interface (requiring administrator/root privileges). For each captured packet, it:

1. Extracts and engineers packet-level features into a DataFrame row.
2. Transforms this single-packet DataFrame using the loaded fitted_preprocessor.
3. Feeds the transformed features to the model's decision_function to get an anomaly_score.
4. Compares the score to the chosen_threshold.
5. Prints an alert if an anomaly is detected.

I initially observed only "Normal traffic" output. To *force* the detection of anomalies for demonstration, I explicitly lowered the chosen_threshold in realtime_monitor.py (e.g., to 0.00 or more negative values). Running a local Nmap scan (nmap -sT 127.0.0.1) in a separate terminal then generated a burst of traffic. With the adjusted threshold, my system successfully detected anomalies during this scan, demonstrating its ability to flag unusual patterns even with the simplified real-time feature extraction. I resolved a recurring minute_of_hour column missing error by consistently ensuring that column was present and not dropped before transformation in realtime_monitor.py, mirroring the fix applied in evaluation.ipynb.

7. Conclusion and Future Work

Through this project, I successfully developed a functional Network Anomaly Detection System capable of real-time monitoring and anomaly detection. I gained practical experience across the entire machine learning pipeline, from data acquisition and complex feature engineering to model training, evaluation, and simplified real-time deployment. The extensive troubleshooting reinforced my debugging skills and understanding of environment and data consistency.

For future development, I plan to:

- Implement a robust, **stateful flow tracking mechanism** in the real-time monitoring component to accurately calculate flow-based features and make real-time detection more meaningful.
- Acquire and integrate a **larger, labeled dataset** (containing both normal and various attack types) to train a more generalized model and perform comprehensive quantitative evaluation (Precision, Recall, F1-score, ROC/AUC analysis).
- Explore more advanced anomaly detection algorithms, including **deep learning models** (e.g., Autoencoders, LSTMs) for complex temporal pattern analysis.
- Integrate the system with **logging frameworks** or a **Security Information and Event Management (SIEM)** solution for persistent alerting and analysis.
- Consider **containerizing** the application using Docker for easier deployment and scalability.

This project laid a strong foundation for understanding and building practical cybersecurity solutions with machine learning.