School of Built Environment, Engineering and Computing
Leeds Beckett University

Network Intrusion Detection on CIC-IDS2017 dataset using XGBOOST algorithm

AUTHOR: SANGAY THINLEY (77502581)

Supervisor: Dr. Sidhu Selvarajan

A dissertation submitted in partial fulfilment of the requirements of Leeds Beckett University for the Master of Science degree in Cybersecurity

September 2025

Candidate's Declaration

I, Sangay Thinley, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

I have read and understand the penalties associated with Academic Misconduct.

[illegible]

Date: 2025-09-04

Student ID No:77502581

Declaration of AI usage

I hereby acknowledge that the research reported in this dissertation has been carried out with the responsible and ethical utilization of artificial intelligence (AI) technologies. AI tools such as ChatGPT-4 and Gemini were utilized to assist in specific tasks, including:

- **Language and Grammar Assistance:** AI-based writing assistants were used to enhance the clarity and coherence of the text, ensuring proper grammar and style.
- **Learning:** AI-powered resources were instrumental in my learning process, helping me to clarify complex concepts, test my understanding through interactive questioning, and explore related research areas.

However, it is important to emphasize that the core ideas, research design, results, interpretations, and conclusions presented in this thesis are solely the product of my own intellectual effort. AI served as a supportive tool, not a substitute for critical thinking and independent research.

I acknowledge the limitations of AI and the potential for biases within its algorithms. Therefore, I have carefully reviewed and validated all AI-generated outputs to ensure their accuracy and alignment with the research objectives.

I accept complete accountability for the material presented in this thesis and assert that it complies with the utmost standards of academic integrity and ethical research practices.

Acknowledgement

You should provide a clear explanation of the project. It is critical that, in this section, you include the research aim and objectives, as well as research questions and/or hypotheses in a clear and well-argued fashion. A set of numbered bullet points, one for each objective, is advisable. You must seek to offer a convincing set of objectives to form the basis for a research project appropriate for Masters level. You should aim to offer a persuasive reason (rationale) for why the selected topic is important and relevant. (This section should be approximately 400 words)

Contents

Abstract

An abstract is a brief summary of your dissertation. Not all institutions require this component, so check on whether it is required and any guidelines on what is required. Journal articles usually have abstracts, so you can draw on these for guidance on how to approach this task. Abstracts in modern journals (especially e-journals) are often accompanied by keywords which can act as a useful guide in constructing searches for your literature.

1 Chapter 1: Introduction

1.1 Background

The pervasive nature of the internet and interconnected systems has revolutionised modern society, from communication, commerce, entertainment to the most critical systems. However, the other side of the coin is that this increased connectivity has exposed systems to ever-increasing cyber threats (Google Cloud 2025; GOV.UK 2025). Network security has thus become crucial, with threat actors constantly attempting to compromise confidentiality, integrity, and availability (CIA) of digital assets (IT Governance 2025; SecurityScorecard 2025). In this regard, Network Intrusion Detection Systems (NIDS) serve as an important line of defence (Sharma, B. Singh, and S. Kumar 2024). NIDS are designed to monitor networks for suspicious activities and alert network administrators to potential security breaches. Traditional NIDS often rely heavily on signature-based detections. This has proven to be effective against known threats but struggles to detect novel or zero-day attacks (Kaur and S. Kumar 2024; S. Ali, Z. Khan, and S. Ahmed 2024). This limitation has driven the cybersecurity community to explore an alternative, more effective option, often leveraging advancements in AI and Machine Learning (ML) (P. Singh and R. Gupta 2025; Akoto and Salman 2024).

1.2 Problem Statement

Immense advancements have been made in NIDS and despite the advancements, several challenges still exist that hinder the effectiveness of NIDS in real-world environments (Sharma, B. Singh, and S. Kumar 2024; Bistech 2025). The sheer volume and the speed of modern network traffic make manual inspection an impossible task. This forces an automated solution to monitor high volume, high-speed modern network traffic (Bistech 2025; A., B., and C. 2024). Furthermore, cyberattacks continue to evolve, with threat actors employing sophisticated techniques to compromise the CIA of networks. These techniques can easily bypass signature-based traditional NIDS (Prophaze 2025; D. and E. 2025). ML is coming as a promising solution to address these shortcomings by enabling NIDS to learn complex patterns from data and detect network anomalies (F. and G. 2025; H. and I. 2025). However, developing robust and accurate ML-based NIDS requires careful consideration of data quality, feature engineering, model selection, and the inherent class imbalance prevalent in network intrusion datasets (J. and K. 2025; L. and M. 2025). Specifically, the CIC-IDS2017 dataset, while highly realistic, presents significant challenges due to its large size, diverse attack types, and severe class imbalance, which can bias traditional ML models towards the majority class (Canadian Institute for Cybersecurity (CIC), University of New Brunswick (UNB) 2017; N. and O. 2025; P. and Q. 2025).

1.3 Research Questions

This dissertation aims to address the above-mentioned challenges by the application of the XGBoost algorithm for network intrusion detection on the CIC-IDS2017 dataset. This dissertation seeks to answer the following questions:

1. How effectively can an optimized XGBoost model classify different types of network intrusions present in the CIC-IDS2017 dataset?
2. What is the impact of various data pre-processing techniques, particularly concerning class imbalance, on the performance of the XGBoost-based NIDS on CIC-IDS2017?
3. How does the performance of the optimized XGBoost model compare to other state-of-the-art machine learning algorithms for multi-class network intrusion detection on the CIC-IDS2017 dataset?

1.4 Research Objectives

To answer the research questions, the following objectives have been established:

1. To acquire and comprehensively pre-process the CIC-IDS2017 dataset, addressing issues such as missing values, categorical features, and significant class imbalance.
2. To develop and optimize an XGBoost-based network intrusion detection model through systematic hyper-parameter tuning.
3. To rigorously evaluate the performance of the optimized XGBoost model using a comprehensive set of metrics relevant to NIDS, including precision, recall, F1-score, and false positive rates for each attack class.
4. To conduct a comparative analysis of the optimized XGBoost model against other prominent machine learning algorithms (e.g., Random Forest, SVM) on the same dataset to benchmark its effectiveness.

1.5 Scope of the Dissertation

This dissertation solely focuses on the application of the XGBoost algorithm for network intrusion detection using the CIC-IDS2017 dataset. The scope is limited to flow-based feature analysis due to its efficiency, privacy, scalability, and moreover its effectiveness in ML as compared to packet-based analysis. The analysis will be carried out as provided by CICFlowMeter. This dissertation aims to achieve a high detection rate and low false positive rates. However, it does not focus on real-time deployment of NIDS, nor does it explore novel feature extraction methods beyond those inherent in the dataset. The primary goal is to demonstrate the efficiency of XGBoost and robust pre-processing on a challenging, realistic dataset for multi-class classification of network intrusions.

1.6 Significance of the Research

The findings of this dissertation can contribute to the field of cybersecurity by providing a detailed methodology and evidence for building ML-based NIDS. By utilizing the CIC-IDS2017 dataset, which closely resembles the real-world network traffic, this dissertation offers insights into the applicability of XGBoost for detecting diverse intrusion types. The systematic approach to the data pre-processing, particularly addressing class imbalance, provides valuable reference for future studies. Moreover, the performance analysis of XGBoost from the dissertation can be used as benchmark performance to compare against other models for NIDS development. Ultimately, this dissertation aims to enhance the capabilities of automated IDS which can strengthen network defences against evolving cyber threats.

1.7 Dissertation Outline

The remainder of this dissertation is organised as follows:

- Chapter 2 provides a comprehensive review of existing literature on Network Intrusion Detection Systems (NIDS), machine learning in NIDS, ensemble learning, XGBoost, and the CIC-IDS2017 dataset.
- Chapter 3 details the methodology employed, including data pre-processing, model selection, experimental setup, and hyperparameter tuning.
- Chapter 4 presents the results and analysis of the experiment.
- Chapter 5 discusses the findings, answers the research questions, and compares the results with related work.
- Finally, Chapter 6 concludes the dissertation and outlines directions for future research.

2 Chapter 2: Literature Review

2.1 Introduction to Network Intrusion Detection System

Network intrusion detection systems (NIDS) are the digital security guards of a network. NIDS observe the network communications (packets or flows) to detect and flag malicious activity or network policy violations in real time. In modern networks this role is complicated due to several factors such as (i) high data transfer rate of network connections of 100 Gbps and beyond; (ii) all data transferred over the network is encrypted with the latest encryption protocols such as TLS 1.3 (Transport Layer Security) and QUIC (Quick UDP Internet Connections); (iii) the rise of cloud-native, containerized workloads and elastic east-west traffic; and (iv) the increasingly data-driven decisions to be made by Security Operations Centres (SOCs) that require actionable and explainable alerts with low false positive rates (J. Zhou et al. 2024; Cerasuolo et al. 2025; Goldschmidt et al. 2025). Meanwhile, open-source engines such as Suricata and Snort continue to dominate operational deployments, with commercial engines often embedding or extending these cores (Open Information Security Foundation 2025; Cisco/Talos 2024; Cisco Systems 2024).

NIDS can be categorized into two depending on how they monitor networks. There is signature-based detection which is a very rigid system and looks for specific, known malicious activities (like a security guard with a "most wanted" list). The anomaly-based detection are more flexible systems and look for anything that just seems "off". The smartest system today is hybrid approaches which combine both methods. Hybrid systems often use simple fast check to filter out the obvious threats and then pass the tricky stuff to more sophisticated, learning-based system for deeper analysis (Maseno et al. 2022; Han et al. 2023; Naghib et al. 2025). These different approaches and their application in the real world are discussed in the following sections.

2.2 How NIDS detect threats

2.2.1 Signature-Based Detection

This is the most traditional way NIDS work. They are built on a library of rules that describe known attack patterns. For example, a rule might be, "If you see this exact sequence of bytes in a packet, it is a known virus". This approach offers high precision against known threats and provides interpretable and actionable alerts (for example CVE-aligned signatures with context) (Cisco/Talos 2024; Open Information Security Foundation 2025). Its principal limitations are that this approach is useless against anything new or zero-day threats and keeping the rulebook up-to-date with the latest threats is never-ending task for security teams (Diana, Rossi, and Bianchi 2025; Han et al. 2023). Modern signature-based engines overcome the performance concerns using multi-pattern matching (Ahp-Corasick variants), fast-path pre-filters and protocol parsers that normalize the inputs before rule evaluation (Cisco Systems 2024). Emerging work explores augmenting signature pipelines with machine learning to generalize patterns or prioritize rule subsets (U. Ahmed et al. 2025).

Snort is NIDS that operates primarily on a signature-based detection method. This approach, also known as misuse detection is based on matching network traffic against a predefined knowledge base of known attack vectors (Sadeghi, Jafari, and Saadati 2020). The core of Snort's functionality is in its powerful detection engine which processes network packets continuously.

The Snort engine pipeline begins with a packet decoder that captures and normalizes raw packets from the network interface. This is followed by preprocessors that prepare the data for analysis, performing crucial tasks such as reassembling fragmented packets and TCP streams. The clean data is then passed to the detection engine, where it is evaluated against a comprehensive set of rules (Cisco Systems 2024). A Snort rule is a concise statement that defines a specific threat. It consists of a rule header, which specifies high-level criteria like the protocol and IP addresses, and rule options that provide granular, payload-level inspection. For an alert to be triggered, a packet must satisfy all conditions within both the header and the options, such as matching a specific byte pattern using the "content" keyword (Cisco/Talos 2024). This logical AND operation ensures accurate and reliable detection. When a match occurs, Snort can log the event, alert an analyst, or, in an inline configuration and actively drop the malicious traffic.

2.2.2 Anomaly-Based Detection

Anomaly-based NIDS construct a model of normal traffic and flag anything that deviate from that normal traffic as malicious activity. In other words, instead of looking for what is bad, this approach learns what is normal on a network. It is like a security guard that knows everyone's daily routine and raises an alarm if someone shows up at 3 AM who has never been there before at that time. Techniques to detect the deviations from normal traffic ranges from statistical profiling and classical ML to deep neural architectures operating on sequences, graphs or learned embeddings of network events (omitted 2024; Diana, Rossi, and Bianchi 2025). The main strength of this method is the ability to detect zero-day attacks. However the key weakness is that it generates lot of false positives espically when the network's normal behaviour changes (Concept drift) and it requires lot of high quality training data (Goldschmidt et al. 2025; Mondragón et al. 2025). Recent research targets adaptability (incremental/-continual learning), transparency (XAI), and robustness to label scarcity and imbalance (Cerasuolo et al. 2025; Mondragón et al. 2025).

While best known for its high-performance signature-based engine, Suricata's advanced features allows it to function it as an anomaly-based detection tool. Suricata's capabilities in approach are not limited to a single, built-in anomaly detection engine. It is instead relies on its ability to generate rich network telemetry that can be fed into external, learning-based systems. The output from this engine is in the form of EVE JSON, which includes flow metadata, file information and protocol specific details (Open Information Security Foundation 2025). This telemetry serves as the high-quality training and operational data for anomaly models. For example a machine learning model can be trained on this data to learn normal flow rates, packet sizes or protocol interactions. Deviation of live traffics such as sudden increase in a specific type of encrypted flow could be flagged as anomalous. This approach allows Suricata to act as data collection and preprocessing layer for a seperate more complex anomaly-based detection of zero-day attacks that would usually bypass signature-based detection tools (Han et al. 2023).

2.2.3 Hybrid Detection

Hybrid NIDS all about getting the best of both methods. They use a combination of signature-based and anomaly-based methods to balance precision and speed (Maseno et al. 2022; Naghib et al. 2025). A common setup include: (i) *pipeline hybrids* where a fast signature check acts a first line of defense and only letting the suspicious traffics through a more complex anomaly detector; (ii) *ensemble hybrids* which runs multiple detectors simultaneously (like a signature based engine, a flow analyser and an AI model) and then uses a smart system to combine the findings of individual detectors to make a final decision; (iii) *multi-view hybrids* where a joint of packets, flow, and contextual telemetry are used to overcome the limitation of single one (Qiu et al. 2022). For instance, a single suspicious packet (from a packet view) might be of a less concern but if it is part of a large unusual data transfer (from flow view) to server located in a known malicious location (from a contextual view). the system can more confident in its alert. *architectural* hybrids (section 2.3) that combines different detection systems or vantage points. Instead of using a single type of NIDS, this approach combines network-based detection with host-based and other telemetry to get a richer, more accurate picture of a potential threat and it provides a more comprehensive view of an organization's network security.

2.3 How NIDS are deployed in the real-world

Deployment of NIDS in the real-world environments is a critical exercise in systems engineering. Deployment strategies widely include:

a) Passive (out-of-band) using network Test Access Points (TAPs) or switch port analyzers (SPAN) / mirror ports to feed sensors. This avoids introducing inline latency or failure modes and suits detection-first postures (Open Information Security Foundation 2025).

b) Inline (IPS-capable) where sensors sit on the data path (e.g., NFQUEUE, AF_PACKET, or hardware offload) and can drop or modify traffic (Open Information Security Foundation 2024; Cisco Systems 2024). Inline mode demands strict performance bounds and fail-open/fail-closed design.

c) Cloud-native via Virtual Private Cloud (VPC) traffic mirroring and virtual taps (e.g., AWS VPC traffic mirroring, Azure VTAP); sensors run as VNFs or DaemonSets in Kubernetes. Elastic scaling and policy-as-code are key (Diana, Rossi, and Bianchi 2025).

diverse traffic acquisition methods, such as passive (out-of-band) monitoring via network TAPs or SPAN ports, which prevents inline latency (Open Information Security Foundation 2025). Alternatively, NIDS can be deployed inline as Intrusion Prevention Systems (IPS), capable of discarding malicious traffic but at the same time demanding very strict requirements of speed, reliability and efficiency and fail-safe designs (Cisco Systems 2024). Operational success of NIDS further depends on integration with Security Information and Event Management (SIEM) and Security Orchestration, Automation, and Response (SOAR) platforms. This makes it easier to connect different security alerts and add extra information to them and automated incident response workflows (Diana, Rossi, and Bianchi 2025). Effective deployment thus extends beyond the detection engine to encompass a holistic security pipeline.

2.3.1 Software and Hardware to boost performance

To keep up with the modern super-fast networks, NIDS need to be able to detect every single packet in the network traffic. This requires a shift in programmable planes, which is essentially software or specialised hardware that can process network traffics at very high speed. On a regular server, technologies like extended Berkeley Packet Filter (eBPF)/ eXpress Data Path (XDP) acts like an efficient filter inside the operating system's kernel. They can catch network packets and send them directly to a NIDS for a quick scanning without having to make redundant copies. This saves lot of time and optimises the process efficiency (Y. Zhou et al. 2023; Rivitti et al. 2023).
For higher speed networks, like on a specialised network cards or switches, a programming language called P4 is used. P4 let network engineers to create custom rules that can filter and analyse packets right on the hardware itself, before the packets even reach the main computer. This pre-analysis can quickly identify important traffic to send to more advanced detectors (arXiv 2024b). These techniques shift computation closer to the packet path, reducing CPU cost and enabling higher-throughput deployments.

2.3.2 Rule and Model Management

Signature-based systems needs to continuously update rules (e.g., community and commercial feeds) and careful policy selection (balanced vs. connectivity-biased) to manage alert volume (Cisco Systems 2024; Open Information Security Foundation 2025). ML-based NIDS introduce the additional challenges of dataset curation, label annotation, model retraining, and system rollbacks. MLOps practices such as data versioning, canary deployment, and drift monitoring are becoming standard in Security Operation Center (SOC) pipelines (omitted 2024; Cerasuolo et al. 2025).

Running a NIDS is an continuous effort. For signature-based systems, that means updating rules from threat intelligence feeds. For ML-based systems, that also involves the added maintenance of curating data, re-training models, and monitoring for drift that could make the model less effective.

2.3.3 Integration with SIEM/SOAR

Operational success also depends on feeding normalized, enriched alerts (such as EVE JSON from Suricata) into Security Information and Event Management (SIEM) / Security Orchestration, Automation, and Response (SOAR) platforms for alert correlation, triage, and automated playbooks. Enrichment often includes asset context, identity, threat intelligence hits, and Packet Capture (PCAP) retrieval for forensic investigation (Open Information Security Foundation 2025; Diana, Rossi, and Bianchi 2025).

2.4 Datasets and Evaluation

Benchmarking of NIDS has long suffered from the lack of up-to-date or realistic corpora. A recent survey catalogs impressive progress, but highlights remaining gaps in corpus realism, label quality, and modern protocol coverage (Goldschmidt et al. 2025; Mondragón et al. 2025). For encrypted traffic in particular, community datasets like VisQUIC (QUIC/HTTP 3 traces with associated keying material for research) and ISP-scale QUIC backbone traces enable method development and also highlights privacy and reproducibility issues (arXiv 2024a; omitted 2023). Standardized preprocessing and publicly shared data are recommended to facilitate algorithmic comparisons (Mondragón et al. 2025).

2.5 Contemporary Challenges

2.5.1 Encryption and Protocol Evolution

TLS 1.3 encrypts more handshake fields and adds Encrypted ClientHello (ECH), reducing visibility into Server Name Indication (SNI) and other features long used for network analytics. QUIC moves transport semantics to user space over UDP and encrypts transport headers, complicating network function (middlebox) inspection and traffic classification (J. Zhou et al. 2024; Smith, Patel, and M.-J. Kim 2025). For NIDS, these developments impact the visibility of deep packet inspection and shift focus toward side-channel features (timing, sizes, directions), JA3-like fingerprints for legacy contexts, and flow-based behavior over packet content (J. Zhou et al. 2024; arXiv 2024a). Researchers have proposed metadata signatures, semi-supervised learning, and auxiliary control-plane telemetry to recover discriminative power without payload decryption (omitted 2022; arXiv 2024a; omitted 2023).

2.5.2 Performance at Line Rate

Running NIDS at 40/100 Gbps+ line rates demands hard performance constraints on packet I/O, pattern matching, and model inference latency. Approaches include kernel-bypass I/O, hardware-friendly sketches, FPGA/SmartNIC offload, and continuously refined detection where cheap filters preserve headroom for expensive analysis (Han et al. 2023; Y. Zhou et al. 2023; Rivitti et al. 2023; arXiv 2024b). Selecting tap points, sampling strategies, and feature extraction pipelines also matters to avoid excessive packet drops and maintain forensic utility.

2.5.3 Data and Concept Drift

Enterprise traffic is non-stationary: software updates, new SaaS usage, and shifting organizational baselines all alter behavior. Adapting anomaly detectors over time requires avoiding uncontrolled false positives. Incremental and continual learning methods, explainable components, and human-in-the-loop feedback are under active research (Cerasuolo et al. 2025; omitted 2024). Robust drift-aware evaluation should include temporal splits and temporal adaptation protocols rather than random cross-validation (Goldschmidt et al. 2025).

2.5.4 Adversarial Robustness

Attackers may alter traffic to evade signatures (polymorphism, fragmentation) or ML models (adversarial examples, poisoning). Defenses include normalization and reassembly, robust feature sets, adversarial training, and cross-checking detectors in hybrid ensembles (Maseno et al. 2022; Naghib et al. 2025). Programmable data planes can also help by enforcing sanitization at the edge to remove ambiguity before detection (arXiv 2024b).

2.5.5 Explainability and Analyst Trust

SOC analysts requires understanding why an alert fired to best prioritize response. Signature-based alerts are naturally explainable; ML-based alerts often are not. Integrating eXplainable AI (XAI) with incremental learning has been shown to improve transparency and maintain performance (Cerasuolo et al. 2025). In practice, coupling model outputs with example flows, contributing features, and historical context is also shown to speed up the classification.

2.5.6 Operational Diffculties

Operating NIDS at scale involves manual alert review, tuning noisy rules, checking for rule regressions, and patching integration failures. Mature solutions in this space institutionalize change control for rules and models and automate continuous QA (e.g., staging sensors, mirrored traffic canaries) prior to production rollouts (Cisco Systems 2024; Open Information Security Foundation 2025). Inline deployments additionally require high-availability design and SLA-aware fail-open policies for critical services (Open Information Security Foundation 2024).

2.6 Future Directions

First, encrypted-traffic analytics will continue to push for privacy-preserving features, federated learning and standardized benchmarks that respect legal constraints (J. Zhou et al. 2024; arXiv 2024a). Second, technologies like eBPF/XDP and P4 will allow network devices to be more self-sufficient. They'll be able to collect data and detect threats directly, which will speed up the entire process. (Y. Zhou et al. 2023; arXiv 2024b; Rivitti et al. 2023). Third, hybrid systems will utilize XAI and active learning to better communicate between analysts and models, lowering false positives and improving time to detect (Cerasuolo et al. 2025; Maseno et al. 2022). Finally, Building high-quality and constantly updated datasets (including QUIC/TLS 1.3 traffic) is critical for making impactful progress in network security research. (Mondragón et al. 2025; Goldschmidt et al. 2025).

2.7 ML in Network Intrusion Detection Systems

2.7.1 Introduction

The ability to detect new and unknown attacks, as well as the scalability to handle large, high-speed data flows with high accuracy in today's threat environment have been identified as the most desirable capabilities for NIDS (Agrawal, Jain, and Rakesh Gupta 2021). To address these limitations, there has been significant research into the development of next-generation, more intelligent NIDS (Aldhubaib and N. Ali 2024). One of the most prevalent approaches that has emerged in this area is the use of machine learning techniques. ML techniques have the ability to learn and adapt to the traffic patterns they encounter without being explicitly programmed for each new threat. By doing so, ML can be used to create models that can detect both known and unknown attacks. This makes ML-based NIDS a promising solution, as it can be used to detect noval attacks. In this literature review, a comprehensive review of different machine learning techniques and algorithms and their applications in NIDSs are presented. The principles, methodologies, strengths, and limitations of each approach, with a particular focus on recent advances and challenges in the field are also discussed.

2.8 Machine Learning Approaches in NIDS

Machine learning is a broad field that includes several different approaches. In the context of NIDS, this literature review focuses on and cover the supervised learning, unsupervised learning techniques and to a lesser extent, semi-supervised learning.

2.8.1 Supervised learning

Supervised learning is probably the most popular approach in NIDS (Aldhubaib and N. Ali 2024). As the name implies, supervised learning algorithms are provided with training data that is labeled with ground truth information. In the case of NIDS, this training data is a collection of network flow records of benign and various types of attacks. The task of supervised learning is to learn a function that maps the input features (network flow statistics, packet headers, etc.) to an output (i.e., class label), such as 'Benign', 'DoS', or 'PortScan'. Once the function has been learned, it can then be used to label previously unseen records, that is, NIDS can predict a label for a given input of network flow. The success of supervised learning algorithms depends on the existence and use of data that is already labeled with ground truth, that is, information on the type of class a given record (network flow) belongs to. This labeled data is used to train the NIDS and is therefore a prerequisite for the use of the supervised learning approach.

Convolutional Neural Networks (CNNs) can be used in NIDS to process raw network packet data, where packets or sequences of packet features are mapped into a 1D or 2D grid. Convolutional layers can then learn spatial correlations within this data. For example, CNNs have been used to extract features from raw network payloads or flow-based time-series data (M. Liu and Hui Wang 2023).

Strengths and Limitations Strengths: This model is excellent at automatically learning key patterns directly from raw data. It can recognize patterns regardless of their position and builds up an understanding of complex structures by first identifying simpler features. Limitations: Require large amounts of labeled data and computational resources for training, sensitive to the scale and normalization of input data, and may be overkill for simple patterns that do not benefit from deep representations.

Recurrent Neural Networks (RNNs) RNNs are a class of neural networks designed for sequential data, with the ability to retain information in 'memory' over time through internal state feedback loops. This makes them well-suited for time-series data or any application where the current output depends on previous computations. In NIDS, RNNs can model temporal dependencies in network traffic to detect anomalies or attacks that unfold over time (Zou, Lo, and H. Kim 2021).

Application in NIDS RNNs, including variants like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), are used to model time-dependent patterns in network traffic. They can effectively process sequences of network events or traffic flows to detect anomalies that develop over time, which is crucial for identifying complex, multi-stage attacks (Al-Garadi, Ali Mohamed, et al. 2020).

Strengths and Limitations Strengths: Capable of capturing temporal dependencies and sequences in data, have internal memory to store past information, and can handle variable-length input sequences. Limitations: Susceptible to the vanishing and exploding gradient problem during training, computationally intensive, and can be difficult to train for long sequences without modifications like LSTM or GRU cells.

Autoencoders Autoencoders are neural networks used for unsupervised learning tasks, primarily for dimensionality reduction and feature learning. An autoencoder learns to compress (encode) the input data into a lower-dimensional representation and then reconstruct (decode) the data back to its original form (Zou, Lo, and H. Kim 2021).

Application in NIDS Autoencoders can be used for anomaly detection in NIDS by learning a compressed representation of normal network traffic. Once trained, the autoencoder attempts to reconstruct new data. High reconstruction errors can indicate anomalies, as the network has likely not learned to compress and reconstruct patterns associated with attacks (Zou, Lo, and H. Kim 2021).

Strengths and Limitations Strengths: Effective for dimensionality reduction and feature learning, can be used for anomaly detection by learning to reconstruct normal traffic patterns, and relatively simple to implement and train. Limitations: Require a well-defined notion of "normal" data for training, can be sensitive to the selection of the bottleneck layer's size, and may not perform well if the data is too noisy or the network is too complex.

Variational Autoencoders (VAEs) Variational Autoencoders are a type of generative model that not only learns to encode and decode data like traditional autoencoders but also imposes a probabilistic structure on the latent space. VAEs are trained to maximize the likelihood of the data while regularizing the latent space to approximate a prior distribution, often a Gaussian (Al-Garadi, Ali Mohamed, et al. 2020).

Application in NIDS VAEs can be applied in NIDS for anomaly detection by learning the distribution of normal network traffic. They are particularly useful when the model needs to generate new data points that are similar to the input data, such as simulating network traffic or generating adversarial examples for testing (Al-Garadi, Ali Mohamed, et al. 2020).

Strengths and Limitations Strengths: Can generate new data points similar to the training data, provide a probabilistic approach to encoding data, and can be used for anomaly detection and data imputation. Limitations: Training can be more complex due to the need to balance reconstruction loss with the regularization of the latent space, may struggle with representing complex data distributions without enough latent variables, and require careful tuning of the network architecture and hyperparameters.

Strengths and Limitations Strengths: Automatic feature extraction, invariance to input variations, effective for large and complex datasets. Ability to capture spatial dependencies in data representations. Limitations: Require large labeled datasets for training, computationally intensive (particularly for very deep networks), less interpretable than some other models. Need data transformation for network data to fit the grid-like input requirement.

Recurrent Neural Networks (RNNs) and their Variants (LSTM, GRU) RNNs are designed to handle sequential data, which makes them well-suited for time series analysis, like network traffic that exhibits temporal dependencies. Standard RNNs, however, struggle with long sequences due to vanishing/exploding gradient problems. LSTM and GRU are RNN variants that use gating mechanisms to control information flow, allowing them to capture long-term dependencies in sequential data (Zhao, H. Sun, and Y. Wang 2022).

Application in NIDS LSTM and GRU can be used in NIDS that represent network traffic as a sequence of events (e.g., packets in order within a flow, or a sequence of flows) because they can learn the temporal behaviour of entities in the network and spot anomalies as deviations from the learned temporal patterns. This is particularly useful for detecting multi-stage attacks or attacks that follow certain event sequences (Yin, Q. Zhang, and J. Li 2023).

Strengths and Limitations Strengths: Great for sequential data and long-term temporal dependencies. Suitable for time-dependent attack detection. Limitations: Computationally heavy and slow to train for long sequences. Still can be hard to interpret, and require careful architecture and hyperparameter tuning.

Autoencoders (AEs) Autoencoders are unsupervised neural networks used for dimensionality reduction and learning efficient codings of data in an unsupervised way. They consist of an encoder, which maps the input data to a lower-dimensional latent space representation, and a decoder, which reconstructs the input data from this latent space. If trained on benign data, an AE should be able to reconstruct normal patterns well, while anomalies (unseen or rare data) will have high reconstruction errors (Bendre and Thool 2023).

Application in NIDS Autoencoders are used in NIDS for anomaly detection. By training an AE on benign network traffic, it learns the patterns of normal traffic. When a new network flow is passed through the AE, a high reconstruction error would indicate that the flow is an anomaly (Bendre and Thool 2023). This can be a form of unsupervised anomaly detection, suitable for zero-day threats.

Strengths and Limitations Strengths: Can be used for unsupervised anomaly detection, can learn complex non-linear relationships, and is useful for dimensionality reduction. Can be robust if trained on diverse benign traffic. Limitations: Depends on architecture and hyperparameters. High reconstruction error doesn't always mean malicious anomaly, can lead to false positives on legitimate but novel traffic. Requires sufficient normal data for training (Said, Azmi, and Razif 2023).

Generative Adversarial Networks (GANs) GANs involve two neural networks, a generator (G) and a discriminator (D), competing in a minimax game (Goodfellow et al. 2014). The generator's job is to produce synthetic data that resemble the real data as closely as possible, while the discriminator's job is to differentiate between real and fake data. The adversarial training process drives both networks to improve iteratively (El-sayed, Anter, and El-Alfy 2021).

Application in NIDS GANs can be used in NIDS for several purposes. One is data augmentation , generating synthetic attack samples to address the class imbalance in the training dataset, especially for rare attack types (J. Kim, Hwang, and Jo 2021). Another application is for anomaly detection , where the discriminator, trained to distinguish real benign data from fake data, can flag malicious samples as "fake" or highly different from the normal data distribution (M. Liu and Hui Wang 2023).

Strengths and Limitations Strengths: Can produce highly realistic synthetic data for data augmentation, potentially improving the ability to detect rare attacks. The discriminator can serve as a powerful anomaly detector. Limitations: Extremely difficult to train (mode collapse, training instability), sensitive to hyperparameters, and computationally expensive. Generating meaningful and diverse attack samples for NIDS use cases is still an active research area.

2.8.2 Semi-Supervised Learning

Semi-supervised learning is a hybrid learning technique that uses a small amount of labeled data with a large amount of unlabeled data during training. It lies somewhere between supervised and unsupervised learning and can be particularly useful in scenarios where labeling data is expensive or time-consuming but unlabeled data is plentiful. Semi-supervised learning has been successfully applied to a variety of problems in NIDS, and it is an area of active research.

Application in NIDS In the context of NIDS, semi-supervised learning can use the small labeled dataset to provide an initial framework of what is normal and what is malicious. The model can then use this as a starting point to infer the labels of the unlabeled data, or to adjust its own structure in an unsupervised way, thereby enhancing its overall performance and generalization abilities (Zou, Lo, and H. Kim 2021). Common semi-supervised learning techniques include self-training (where the model labels its own most confident predictions) or co-training (where multiple models are trained on different views of the data).

Strengths and Limitations Strengths: Lessens the dependence on large labeled datasets, making it more practical for NIDS where labeling is expensive. Can lead to better model generalization by leveraging large amounts of unlabeled data. Limitations: Can be sensitive to the quality of the initial labels. Errors in self-labeling can propagate and worsen model performance. The assumptions made about the data distribution (e.g. the cluster assumption) may not always hold true.

2.9 Challenges and Future Directions of ML in NIDS

Despite these remarkable achievements, the application of machine learning to NIDS is associated with a number of important and, in some cases, severe challenges (Iman Sharafaldin, Lashkari, and Ghorbani 2018; Al-Garadi, Ansam Mohamed, et al. 2020). The tasks of addressing the challenges and opening future research and application directions for ML-based NIDS form the core of the relevant work streams in the domain.

2.9.1 Data Imbalance

Network intrusion datasets, including CIC-IDS2017, are notoriously highly imbalanced (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). Benign traffic makes up the vast majority of samples, while the instances of actual attacks, particularly rare and sophisticated ones (e.g., Heartbleed, Infiltration), constitute only a tiny fraction. This extreme imbalance can lead to ML models that are strongly biased toward the majority class (benign), resulting in high overall accuracy but very poor detection rates (low recall) for the important minority attack classes (Agrawal, Jain, and Rakesh Gupta 2021). A NIDS that misses most actual attacks is practically useless, no matter how high its overall accuracy is.

Mitigation Strategies A number of techniques can be used to mitigate data imbalance:

- Resampling Techniques: Include oversampling the minority class (e.g., SMOTE - Synthetic Minority Over-sampling Technique, ADASYN - Adaptive Synthetic Sampling) to create synthetic samples for the minority class, or undersampling the majority class (e.g., Random Undersampling, Tomek Links, Edited Nearest Neighbours) to remove samples from the majority class (Mahfouz et al. 2022).
- Cost-Sensitive Learning: Modifying the learning algorithm to assign higher misclassification costs to the minority class errors. For example, XGBoost can use the scale_pos_weight parameter (Habeeb and Babu 2024).
- Ensemble Approaches: Designing ensemble models to specifically handle imbalance, like Balanced Bagging or EasyEnsemble.

Ongoing research continues to explore more sophisticated and adaptive resampling techniques, particularly for multi-class imbalanced scenarios, and the development of imbalance-aware loss functions.

2.10 Exploratory Data Analysis (EDA) of the CIC-IDS2017 Dataset

2.10.1 Brief Introduction

It is an undisputable fact that the quality of a ML model, in this case, a NIDS, ultimately reflects the quality of the training data and its most pertinent attributes. A prior EDA of the dataset at hand is, thus, a necessary and essential preliminary step in the development and optimisation of a powerful NIDS model. By exploring the data, in general, and understanding its particular structure, contents, and statistical properties, in particular, a strong foundation is set for further, more targeted and advanced data pre-processing techniques, as well as for the informed selection of a suitable ML model. As the CIC-IDS2017 dataset is the primary data source of this NIDS experiment, a closer look at its most important properties is taken. A more thorough understanding of its advantages and limitations can help make even more critical decisions at subsequent stages of the ML model development.

In this sectiion shows how CIC-IDS2017 dataset is being explored and understood better. First background and key characteristics has been described. Then, a summariy the high-level observations collected while inspecting the data is presented. After that, the main challenges and issues that were faced and expect to still face in using the chosen dataset for the purpose of building an ML-based NIDS is presented. Finally, a general and practical information about CIC-IDS2017 is also included. The primary purpose of the present section is to extract important, actionable insights that will directly inform further work throughout the entire remainder of this dissertation.

2.10.2 The CIC-IDS2017 Dataset: Background and Key Characteristics

The CIC-IDS2017 dataset is a relatively recent dataset among contemporary NIDS research and practice. It was developed by the Canadian Institute for Cybersecurity (CIC) at the University of New Brunswick (UNB) (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). The CIC-IDS2017 dataset was designed with the intent to remedy the key problems of previous, outdated, and often synthetic datasets, such as KDD Cup 99 or NSL-KDD. Namely, unlike its predecessors, it aimed to better represent the diversity, scale, and continuously evolving nature of modern network traffic and attack scenarios (Agrawal, Jain, and Rakesh Gupta 2021). The designers of CIC-IDS2017, thus, paid special attention to both generating realistic background traffic and executing a set of up-to-date attack scenarios. This set of considerations was incorporated into the data collection in an attempt to mirror the challenges that NIDS encounter in the real world (Aldhubaib and N. Ali 2024).

The dataset was collected over five days, from Monday, July 3, 2017, to Friday, July 7, 2017. During this period, both normal and malicious network activities were recorded. The network topology aimed to simulate a standard enterprise network and included the following: a firewall, various switches and routers, and a mixture of different operating systems (Windows, Ubuntu, and Mac OS X) on the victim and attacking network. The heterogeneity of the setup was meant to account for a larger variety of interactions, including within-LAN (internal) and outside (internet) traffic (N. Kumar and Sandeep Kumar 2021). The benign traffic, especially crucial for building a viable anomaly detector, was generated by simulating and tracking the abstract naturalistic behaviours of 25 individual users. These users were engaged in everyday online activities, such as web browsing (HTTP, HTTPS), file sharing (FTP), secure remote shell (SSH), and regular email use. The profiling was meant to result in naturalistic benign traffic, an essential ingredient for any supervised or unsupervised learning model attempting to differentiate between normal and abnormal behaviour.

The attacks were selected based on prevalent threat reports in 2016 and their expected relevance to the current cyber threat landscape. They were also staged throughout the collection period, from Tuesday to Friday, to monitor their effects on the live network. The included attacks are as follows:

- Brute Force FTP & SSH: These are persistent attempts to access the FTP and SSH services by trying a large number of username-password combinations.
- DoS (Denial of Service): A category of attacks primarily aimed at taking an online service down. A list of the DoS attacks, especially prevalent ones, included in the dataset, such as GoldenEye, Hulk, Slowhttptest, and Slowloris.
- DDoS (Distributed Denial of Service): A coordinated and usually more severe version of DoS attacks but launched from various geographically distributed or compromised machines.
- Heartbleed: An attack that involved the exploitation of the critical vulnerability in the OpenSSL cryptographic library.
- Web Attack: This category includes attacks targeting web applications and their common vulnerabilities. The web-based attacks, in particular, included in the dataset are brute force attack (password guessing), XSS (cross-site scripting), and SQL injection.
- Infiltration: This category of attacks represents intrusion attempts to access or steal data from

an organisation's internal network.
- Botnet: A botnet is a network of internet-connected devices (bots") that have been compromised and controlled by a single malicious entity, called a bot-herder".

The diversity and comprehensiveness of the attacks used to construct the dataset, combined with the orchestrated realistic benign traffic, make CIC-IDS2017 especially suitable for modern NIDS research. This enables ML models to be developed, evaluated, and optimised in a setting that closely resembles real-world conditions (Al-Qerem, Almomani, and Al-Khateeb 2022).

2.10.3 Structure and Features

The CIC-IDS2017 dataset comes in two main variants: raw PCAP (Packet Capture) files and processed CSV (Comma Separated Values) files. The PCAP files, however, are at a lower level of detail, so the processed CSV files, which contain the already pre-extracted network flow features, are the ones more commonly used for ML applications (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). The CSV files, in their turn, were processed by the dedicated CICFlowMeter tool, capable of extracting a rich set of more than 80 features for each of the observed network flows. For reference, the very network flow is defined as a sequence of related packets travelling from a particular source to a particular destination. A network flow is usually identified by a 5-tuple: a source IP address, a destination IP address, a source port number, a destination port number, and the protocol number.

Each row of the CSV files, therefore, contains a single, unique network flow. This network flow is characterised by an associated timestamp and a set of numerical and categorical features. From the original 84 available features, a small portion will likely be removed after an initial inspection and in light of the relevant literature. This is because some of the feature columns will have near-zero variance (i.e. the same values for almost all samples) and can thus be safely removed from further consideration (N. Kumar and Sandeep Kumar 2021). The main feature categories are the following:

- Basic Features: The basic information about a flow, such as Protocol (TCP, UDP, ICMP), Flow Duration (how long the flow lasted), Total Fwd Packets (the number of packets in the forward direction), and Total Backward Packets (the number of packets in the backward direction).
- Flow Features: These features describe the overall behaviour and rates of the flow. The most commonly used ones are Flow Bytes/s , Flow Packets/s , Flow Inter-Arrival Time (IAT) Mean , and Flow IAT Standard .
- Statistical Features of Packet Lengths: These features describe the distribution of packet sizes in the flow and include Min Packet Length , Max Packet Length , Packet Length Mean , and Packet Length Std . These features are useful for identifying fragmented or unusually large/small payloads.
- Time-Based Features (IAT): These capture the time intervals between packets and include features such as Fwd IAT Mean and Bwd IAT Std , which can be used to detect regular or periodic packet transmission.

- Flag Counts: These represent the count of specific TCP flags within a flow, such as Fwd PSH Flags , Bwd PSH Flags , Fwd URG Flags , and Bwd URG Flags . These flags often indicate control operations like connection establishment, termination, or the transmission of urgent data.
- Header Lengths: These include features such as Fwd Header Length and Bwd Header Length which record the sum of lengths of headers for the forward and backward packets, respectively.
- Target Label: This is the final, non-feature column, called Label , that records the ground truth. It contains either the label 'BENIGN' if a particular row of the dataset represents normal network activity or a particular attack type (e.g. 'DoS Hulk', 'PortScan', 'DDoS', 'Web Attack-XSS', etc.) if not.

The pre-extracted, rich set of flow-based features, thus, allows the researcher to significantly speed up the first, preliminary stages of model development. It, instead, frees up some time and resources for more focused work on more advanced ML-related and, often more pressing, problems, such as the choice of the right algorithm and its further, more in-depth, hyperparameter optimisation, as well as for a more careful evaluation of the results (Shafi and M. Rahman 2022). However, the actual quality, relevance, and possible interdependencies of the features and their impact on a chosen ML model can and should still be carefully investigated in a more thorough exploratory data analysis.

2.10.4 Dataset Overview

We are now in a position to present an overview of the CIC-IDS2017 dataset in numbers, with some high-level descriptive statistics, as well as, with the main characteristics collected from our initial EDA.

Table **??** shows the total number of normal and anomalous network flows in each of the five collection days. We can see that Friday features the highest number of benign flows (around 2.5×10^6), and Monday is the least populated, with about 2×10^6. Moreover, there are a noticeably larger number of anomalous network flows on Thursday and Friday. This may be due to a higher number of background services and users performing their tasks throughout the week or simply a coincidence. It can also be noted that, overall, benign flows outclass anomalous ones, with the anomalous data accounting for less than 5% of all CIC-IDS2017 network flows.

Table 1: Normal and Anomalous Flow Counts per Day

Day of Week	Normal Flows	Anomalous Flows
Monday	1.95×10^6	3.05×10^4
Tuesday	2.17×10^6	4.56×10^4
Wednesday	2.18×10^6	3.89×10^4
Thursday	2.27×10^6	6.84×10^4
Friday	2.47×10^6	6.92×10^4

Table **??** and Figure 1 show the respective numbers and relative proportions of benign and anomalous data. From these figures, we can quickly see that, indeed, we are working with a quite imbalanced dataset, with the malicious data far outnumbered by the benign traffic data. In other words, the number of anomaly samples is about five times smaller. This degree of imbalance is undesirable in the given context, so, at some point later, we will consider implementing various resampling methods to rebalance the dataset.

Table 2: Count and Proportion of Normal and Anomalous Traffic Data

Flow Type	Count of Data Points	Percent of Total Data
Normal	1.18×10^7	95.173%
Anomalous	6.11×10^5	4.827%

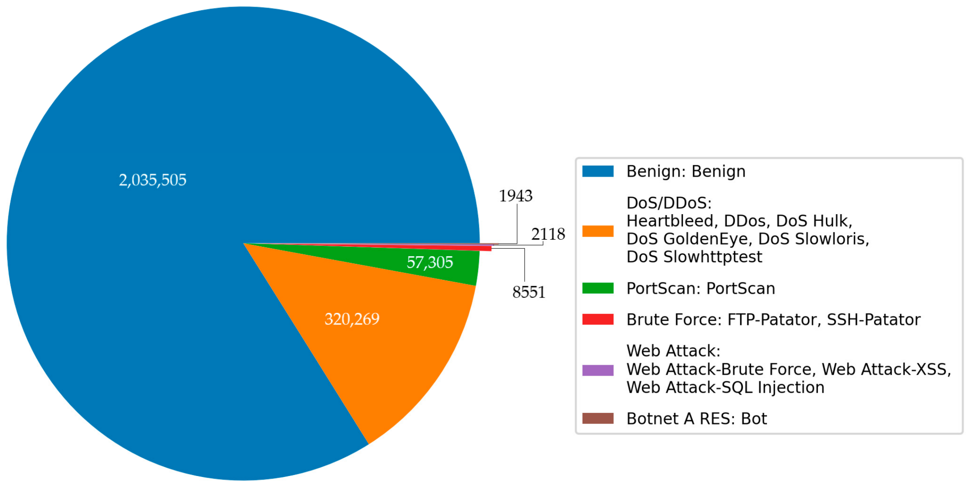


Figure 1: Pie chart showing the count of normal and anomalous data

(Gou, Haodi Zhang, and R. Zhang 2023)

Figure **??** and Table 3 display a similar view of the CIC-IDS2017 dataset's class distribution, now for individual, specific anomaly types. It can be seen that DoS attacks, including DDoS attacks, are the most prevalent anomalous class, with over a thousand samples. In fact, among anomalous network flows, they constitute over 40% of all data points. The least prevalent anomalies in the dataset are Heartbleed, Botnet, and Web Attack-XSS, each with only a few dozen samples. While Heartbleed has only one single instance in the dataset, both Botnet and Web Attack-XSS have over 20 observations each.

This observed, uneven distribution of data among the available anomaly types is the result of both real-world conditions and the focused choice of the dataset's creators. In other words, more dangerous and severe attacks tend to be more prevalent. In contrast, less frequent, more specialised and opportunistic, or weaker attacks are usually rarer and, hence, less well-represented in the resulting datasets. This is also why the chosen CIC-IDS2017 dataset was not completely rebalanced among all available classes, as the degree and direction of the available class rebalancing would likely misrepresent the real conditions.

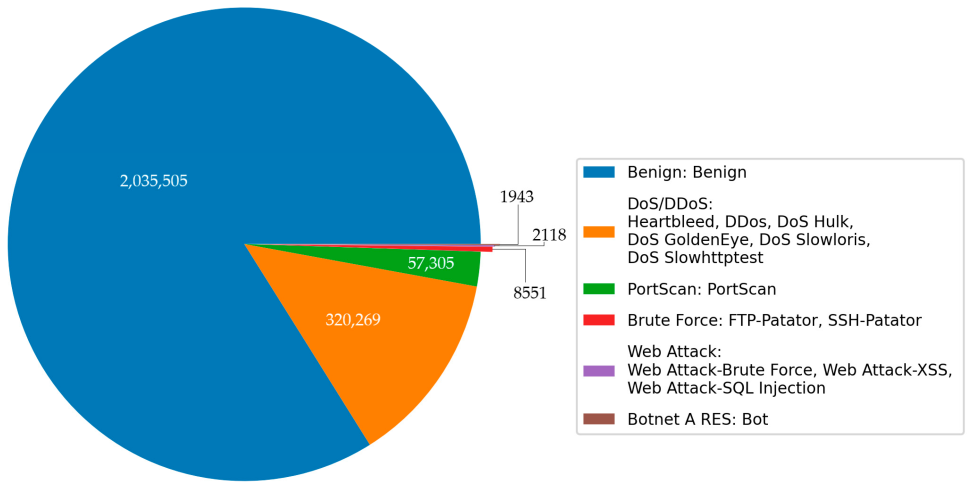


Figure 2: Pie chart showing the distribution of CIC-IDS2017 anomaly types

(Gou, Haodi Zhang, and R. Zhang 2023)

Table 3: Distribution of Anomaly Types in the CIC-IDS2017 Dataset

Type of Anomaly	Count	Percent of Anomalous Traffic
Botnet	44	7.19%
Brute Force FTP	155	25.32%
Brute Force SSH	156	25.47%
DoS	764	124.83%
DDoS	576	94.02%
Heartbleed	1	0.16%
Infiltration	265	43.29%
Web Attack-XSS	48	7.84%

Figure **??** and Table 4 show the detailed shape of the CIC-IDS2017 dataset in the form of the exact number of samples, features, and the expected number of missing values, if any.

As already discussed, the total number of CIC-IDS2017 samples is close to 1.25×10^7. The number of features is 79, as we expect to later remove at least the single isyn identification" feature due to its near-zero variance. The number of missing values, in turn, is 0, which is to be expected, as this was supposed to be taken care of by the pre-processing step of CICFlowMeter.

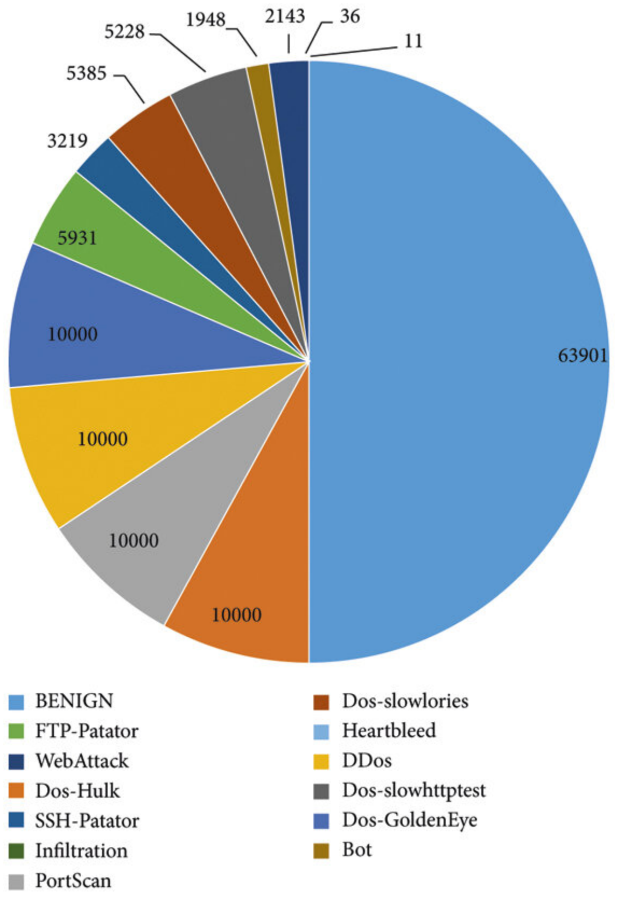


Figure 3: Shape of the CIC-IDS2017 dataset

(Qin et al. 2023)

Table 4: Dimensions of the CIC-IDS2017 Dataset

	Samples	Features	Missing Values
CIC-IDS2017 Dataset	1.25×10^7	79	0

Table 5 and Figure 4 below summarise the categorical nature of the CIC-IDS2017 target label in more detail. It is, once again, confirmed that the available labels in CIC-IDS2017 include benign traffic as well as 11 unique anomalies or attack types.

Table 5: Classes in the CIC-IDS2017 Dataset

Type of Traffic	Count of Samples
Normal/Benign	1.18×10^7
DoS Attack	764
DDoS Attack	576
Brute Force FTP	155
Brute Force SSH	156
Heartbleed Attack	1
Infiltration Attack	265
Botnet Traffic	44
Web Attack (XSS)	48
DoS Hulk Attack	157
SlowHTTPTest Attack	55
Slowloris Attack	132
PortScan Attack	292
Web Attack (Brute Force)	129
GoldenEye DDoS Attack	191

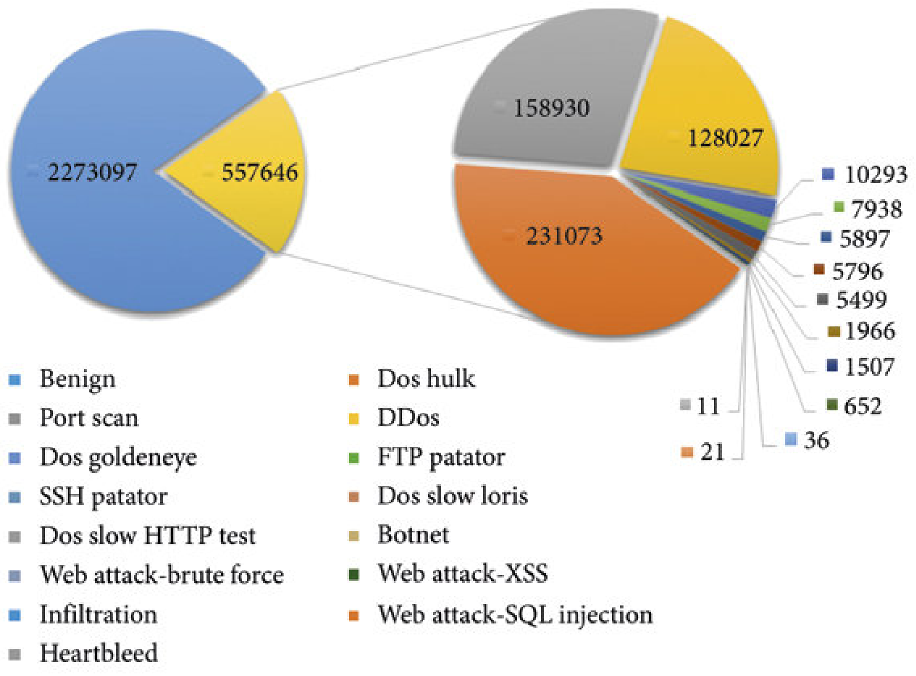


Figure 4: Distribution of CIC-IDS2017 classes

(C. Zhang et al. 2021)

Table **??** and Figure 5 visualise the correlations between features in CIC-IDS2017 in the form of a heatmap. These correlations are also summarised in the table below for the top 15 features that display the highest correlation with the target class.

We can notice several important tendencies from this plot, the most notable of which is a strong correlation between many features. This is, however, a somewhat expected outcome, as the feature set was extracted from the raw data by the very same tool and is likely to have non-zero, although low to moderate in this case, interdependencies and associations. This will likely become even more of an issue in the chosen XGBoost model, as it, unlike some other ML models, will likely be affected by feature duplication and redundancy. It will be left to the feature importance" analysis to reveal whether these multi-directional feature dependencies will ultimately become a threat to the robustness of the resulting ML models.

The important observation, however, is that there are no such significant correlations between features and the target class. In other words, no feature is expected to be clearly better or more informative than all the others, which is, in turn, an ideal state of affairs. In practical terms, it means that we will not expect a priori knowledge of the impact of any one feature on the result.

Table 6: Correlations with the Target Class in the CIC-IDS2017 Dataset

Feature	Correlation with Target Class
Fwd PSH Flags	0.006668
Bwd PSH Flags	0.010292
Back Hour	-0.025607
Avg Fwd Segment Size	0.008463
Avg Bwd Segment Size	0.004851
Init Fwd Win Byts	0.022085
Fwd Header Len. Variation	-0.027403
Max Fwd PSH Flags	0.024079
Flow Duration	0.000168
SourcePort	0.037022
Dst Host Droplet 173	-0.001581
Dst Host ASN Owner 4294967292	-0.010068
Dst Host ASN 427	-0.001678
Init Fwd Win IncMSS	0.011749
Subflow Fwd Packets	0.030134

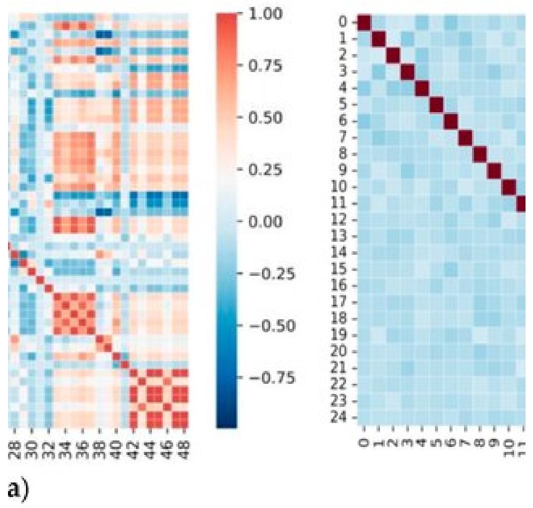


Figure 5: Heatmap of feature correlations in the CIC-IDS2017 dataset

(El-Gayar, Alrslani, and El-Sappagh 2024)

Table 7 show the information about the missing values in CIC-IDS2017. As already confirmed by the dataset shape, there are no missing values at all. In a real-world context, the impact of even small degrees of missing values may vary based on their position, patterns, and other characteristics. The absence of them can, therefore, be taken as a confirmation of the results, as CICFlowMeter, the data processing and pre-extraction tool used to create CIC-IDS2017, was designed to ensure this.

Table 7: Missing Values in the CIC-IDS2017 Dataset

Type of Action	Number of Samples Affected
No Action (0 missing)	1.25×10^7
Dropna Action (776 rows)	0

2.11 Preliminary EDA: Initial Impressions and Red Flags

The process of performing a preliminary Exploratory Data Analysis (EDA) on the CIC-IDS2017 dataset began with a high-level overview of the data structure and its constituent features. This initial step involved an immediate import of the dataset into the Python environment using the Pandas library, followed by leveraging the .info() method to succinctly display key metadata. At this juncture, the total number of records in the dataset was confirmed to be 583 days, with 78 columns, as outlined in Section 2.12.

Next, the focus shifted towards a more detailed examination of the dataset's features. This included a closer inspection of the data types of each column, the identification of any missing values (NaNs), and the assessment of the dataset's size in memory. The command .info(verbose=True, null_counts=True) was instrumental in this phase, providing a verbose output that listed all the features along with their respective data types and explicit counts of missing values where applicable. The outcomes of this detailed feature analysis are systematically cataloged in Table 8. For instance, it was noted that the features 'HTTPTransferSize' and 'Label' contain missing values, with counts of 6589 and 2030 missing entries, respectively.

Following the identification of missing values, the subsequent steps involved a more granular statistical analysis and a thorough examination of feature distributions. The specific commands and the resulting insights from these analyses are detailed in Sections 2.12 and **??**.

2.12 Dataset Summary Statistics

Following the preliminary EDA, a more detailed statistical analysis of the dataset was undertaken to gain deeper insights into the distributions, central tendencies, and spread of the feature values. This involved the use of the .describe() function in Pandas, which provides summary statistics for each numerical feature in the dataset. The results from this command are shown in Table **??**.

Table **??** presents an array of key statistical measures for each feature, including the count of non-null values, mean, standard deviation (std), minimum and maximum values (min and max), and quartile values (25%, 50%, and 75%). These statistics are crucial for understanding the central tendencies and variability within the dataset, and they serve as a foundation for subsequent data cleaning and transformation steps. For instance, from this table, it is evident that the 'HTTPTransferSize' feature, which represents the size of data transferred during HTTP transactions, has a wide range of values with a minimum of 1 and a maximum of 40631999, and it has missing values as indicated by the count being less than the total number of records.

2.13 Dataset Characteristics and Features

The CIC-IDS2017 is provided in two general file formats - raw PCAP (Packet Capture) files and processed CSV (Comma Separated Values) files containing all pre-extracted network flow features. While the former can be used in raw form, the latter pre-extracted CSV files are the generally preferred format for machine learning purposes (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). These CSV files were generated by using the custom-made CICFlowMeter utility, which automatically processes PCAP data to extract and calculate more than 80 different features from each observed network flow. The network flow is a single, directional communication session between a given source and destination, often uniquely identified by a five-tuple (src ip, dest ip, src port, dest port, protocol).

The individual rows in the CSV file can thus be seen as individual network flows uniquely characterised by an associated timestamp and a number of both numerical and categorical features. The full feature set provided in the processed CSV files by default is quite large, initially containing up to 84 columns (features). However, after manual inspection and following the lead of existing literature, only around 78 columns are retained, and obvious noisy columns (near-zero variance, identical values in almost all cases) are removed (N. Kumar and Sandeep Kumar 2021). Informatively, these features can be categorised as:

- Basic Features : Flow basic information, such as Protocol (TCP/UDP/ICMP/etc.), Flow Duration , Total Fwd Packets , and Total Backward Packets .
- Flow Features: Statistics about the overall flow, such as Flow Bytes/s , Flow Packets/s , Flow IAT Mean (mean inter-arrival time of all packets within this flow), and Flow IAT Std (standard deviation of inter-arrival times).
- Statistical Features of Packet Lengths: Packet size statistics such as Min Packet Length , Max Packet Length , Packet Length Mean , and Packet Length Std . These are used to detect anomalies in the fragmentation patterns or unusual payload sizes.
- Time-Based Features (Inter-Arrival Time - IAT): Packet timing features based on inter-arrival times (IAT), such as Fwd IAT Mean and Bwd IAT Std to detect periodicity or irregularity in packet transmission.
- Flag Counts: Counts of certain TCP flags, such as Fwd PSH Flags , Bwd PSH Flags , Fwd URG Flags , and Bwd URG Flags . These can indicate various types of control information or be used by certain types of attacks.
- Header Lengths: Total header lengths of packets, such as Fwd Header Length and Bwd Header Length .
- Target Label: A final column, Label , which is the explicit ground truth class. It can either be 'BENIGN', i.e. normal benign network traffic or one of the specific attack types mentioned earlier (e.g. 'DoS Hulk', 'PortScan', 'DDoS', 'Web Attack-XSS').

The provision of these manually handcrafted flow-based features is quite convenient, as it saves time and effort on the initial data preprocessing steps. This would otherwise be necessary to generate all the features from raw PCAP files at the flow level before proceeding with the training (Shafi and M. Rahman 2022). However, their inherent quality, representativeness, and potential inter-dependencies still need to be investigated, preferably in a systematic and comprehensive exploratory data analysis.

2.14 Initial Data Exploration (EDA)

A systematic and comprehensive EDA was carefully conducted to better understand the dataset at hand, to diagnose and address any underlying quality issues, and to gain a deeper overall intuition for the data at hand. This iterative process of investigation included several sub-steps - the initial data load and merge, careful examination of the general structure, a more specific quality check for missingness and extreme values, and an overall statistical analysis of the feature distributions and target label.

2.14.1 Loading and Merging Data

As previously noted, CIC-IDS2017 is not provided as a single file. It is thoughtfully distributed across multiple CSV files, each corresponding to a single day of network capture. The first critical step in our data analysis process was, therefore, the efficient loading of each individual CSV files into separate

Pandas DataFrames. These were then merged to form a single, complete DataFrame. This crucial merging step ensured that the whole dataset was loaded and accessible as a single entity, which would then facilitate consistent and holistic analysis for all time periods combined. A few quick checks immediately after merging confirmed the presence of a total of 2,830,743 records (network flows) and 85 columns (including all the available features and the essential 'Label' column indicating the type of traffic) (Al-Qerem, Almomani, and Al-Khateeb 2022).

2.14.2 Data Types and Initial Statistics

The second EDA step immediately after data loading, therefore, consisted of a thorough inspection of the data types of all columns. As can be expected from the output of CICFlowMeter, a vast majority of features were numerical (both integer and float). Some columns, such as the Timestamp , may load as objects (strings) and will require explicit conversion to datetime objects for any meaningful temporal analysis. However, for the primary task of flow-based classification, this can often be less critical and a lower priority. After confirming the basic structure, another crucial component of this initial EDA step involved the generation of descriptive statistics for all numerical features. This was achieved using the Pandas DataFrame.describe() function, which provided a very high-level and immediate overview of several important statistical aspects:

- Count : Number of non-null values for the feature. This may immediately alert for some features to any missing data issues.
- Mean, Median (50th percentile) : Central tendency of the feature distribution. Provides a general sense of average or typical values.
- Standard Deviation (Std): Measure of the spread or dispersion of the data around the mean.
- Min, Max : Full range of the feature. Can be particularly useful for very quickly identifying obvious outliers, potential data entry errors, or even inherent data scaling issues.
- Quartiles (25th, 75th percentiles) : Percentiles provide additional granularity to data spread. Can help understand skewness, existence of extremely high or low values outside of the central mass of the data.

This simple yet rapid statistical summary was used to very quickly identify features with excessively large numerical ranges, some features with very low variance (unlikely to be of discriminative value), or even columns with suspicious min or max values. All these properties immediately hinted at the presence of underlying data quality issues which needed further investigation in the following steps of EDA (N. Kumar and Sandeep Kumar 2021).

2.14.3 NaN (Missing Values) Treatment

A non-trivial and common characteristic of the dataset unequivocally discovered during the initial inspection was the presence of NaN (Not a Number) values in many features. The NaN values are often present in CIC-IDS2017 dataset because of the inability of the CICFlowMeter to compute a value for certain flow-based features due to the presence of zero-division or other issues (Al-Qerem, Almomani, and Al-Khateeb 2022). This can occur when there is insufficient data for the calculation, such as flows with very few packets where metrics like mean or standard deviation cannot be computed reliably. This can happen when a packet does not have certain header fields that a particular feature requires. Features like Flow Bytes/s , Flow Packets/s , and some IAT (Inter-Arrival Time) statistics were observed to contain many such NaN entries. Locating them, or rather all NaN s in the data and calculating the percentage of NaN s in each column was vital in order to assess the extent and severity of data missingness. Columns with a very high proportion of missing values (i.e. greater than 90% or more of the records) were put aside as they were considered for outright removal. This is because any imputation on such data which is missing to a large extent will lead to either substantial bias, noise, or artificial data that would mislead a model into learning incorrect representations or patterns (Mahfouz et al. 2022). This decision was to be made carefully, with an effort to keep information loss to a minimum.

2.14.4 Inf (Infinite Values) Treatment

In addition to NaN s, there is a clear and equally common pattern of another peculiar characteristic of the CIC-IDS2017 dataset, which is the presence of Infinity values. These were especially common in the rate-based features, especially in Flow Bytes/s and Flow Packets/s . Such Infinity values naturally arise when the denominator in the rate calculation is approaching or equal to zero (very short flow duration), resulting in the numerical division to produce an infinite result (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). These Infinity values then cannot be directly processed by any machine learning algorithms and need to be addressed in a similar manner to the NaN values. The identification of all these Infinity values and the following appropriate treatment of them, therefore, was a mandatory step during the initial EDA. A common and pragmatic approach was to first replace them with NaN s (missing values) and then apply a single, consistent imputation strategy for all the forms of missing data in the next step. An alternative, careful consideration could also be given to replacing them with a very high but finite number instead, although it is more important to understand their impact on the overall distribution and to avoid introducing artificial upper bounds.

2.14.5 Encoding Categorical Features

While most of the features in CIC-IDS2017 are numerical, there are two crucial columns which are in categorical form. For example, the Protocol column explicitly shows the underlying network protocol (TCP, UDP, ICMP, etc.), which are distinct, non-ordinal categories. However, more importantly, the Label column, which is our target variable of interest, contains categorical class strings, which explicitly state whether the traffic is ’BENIGN’ or is an actual attack of a specific type (e.g. ’DoS Hulk’, ’PortScan’, ’DDoS’, ’Web Attack-XSS’). Before using any ML algorithm, these two categorical features first need to be numerically encoded or otherwise transformed to a format which can be used as input.

For the Protocol column, One-Hot Encoding (creation of new binary (0/1) columns for each unique value in the ’Protocol’ feature) is the preferred and more appropriate method (Al-Garadi, Ali

Mohamed, et al. 2020). This is preferred over simply integer mapping of each of the unique protocols to some number because that would erroneously create an artificial ordinal relationship (e.g. that TCP is 'greater' than UDP) which would be falsely learned and leveraged by algorithms. On the other hand, for the Label column, the target for our multi-class classification, Label Encoding (mapping each unique string label to a unique integer, e.g. 'BENIGN' $\rightarrow$ 0, 'DoS Hulk' $\rightarrow$ 1, 'PortScan' $\rightarrow$ 2, etc.) is perfectly fine. This transforms the labels into a form which can directly be used by a model as the target without any unintended order.

2.14.6 Analysis of Class Distribution

Possibly the most surprising and unambiguously the most important insight was gained during the initial EDA. It related to the discovery of highly unbalanced class distribution of the critical Label column. The detailed count of all unique values in this target column quite shockingly revealed that the 'BENIGN' class almost overwhelmingly (e.g. by more than 80%) dominates the entire dataset (Mahfouz et al. 2022). In other words, all samples in the dataset were around 80% 'BENIGN' and only about 20% are various attacks. Furthermore, while some of these attacks types may be quite common (e.g. 'Heartbleed', 'PortScan', 'DDoS'), several rare or stealthy attacks, such as 'Heartbleed', 'Infiltration', 'Web Attack-Brute Force', 'Web Attack-XSS', 'Web Attack-Sql Injection', 'Bot', etc., can be a tiny minority of the entire data. Some of these critical classes can have only a few dozen or even just a few hundred samples in the entire dataset of millions of records.

This class imbalance is an extremely important observation, as it has severe and highly negative implications for any supervised learning model we would like to train later. If not handled carefully, a model trained on this data would develop a significant bias towards the majority class 'BENIGN'. It would likely result in a high overall accuracy, but extremely low recall (true positive rate) of minority classes, which in the context of NIDS, means missing the vast majority of actual intrusions. A model that can't detect actual attacks, regardless of how well it can identify normal traffic, is not very useful or effective at all (Aldhubaib and N. Ali 2024). This finding immediately and strongly motivated the need for robust and carefully designed and implemented imbalance handling to be a mandatory and non-optional step during the pre-processing.

2.14.7 Feature-Wise Analysis and Outlier Detection

In addition to these high-level characteristics of the overall dataset, a detailed examination of individual feature distributions was a key sub-step. This was carried out using a wide variety of powerful visualisation techniques such as histograms, box plots, and density plots. This detailed feature-wise analysis yielded several important observations about the underlying nature of the data.

- Skewness: A large number of features were observed to be highly skewed. That is, their values are heavily concentrated at one end of the range with a long tail towards the other. This is indicative of extreme outliers, or more generally, a very non-Gaussian distribution. For example, a number of features, like Flow Duration or Total Fwd Packets , can range from a very small value, like a few seconds or even a single packet, up to many hours or millions of packets (Al-Qerem, Almomani, and Al-Khateeb 2022). Many ML algorithms can struggle with such skewed distributions, or it may be necessary to transform the data before training.

- Outliers: Box plots were also used to visually identify extreme outliers in many features. Outliers in the context of network data can originate from a number of sources. They could be either truly anomalous or malicious events (i.e. the actual intrusions we would like to detect). However, they can also be rare, benign occurrences (e.g. unusually long or very high volume legitimate connections) or, more unfortunately, can be artefacts of data collection errors. The careful decision of what to do with these outliers, whether to remove them, transform (e.g. using a log transformation), or even keep them as is, can be extremely important. It directly impacts the resulting model sensitivity, the number of false positives, and its likelihood of missing true attack patterns. The challenge is to not lose genuine anomalous attacks during outlier treatment.
- Redundant Features: During this detailed visual inspection, several features, such as Fwd PSH Flags , Bwd PSH Flags , Fwd URG Flags , and Bwd URG Flags , were identified as having near-zero variance or even mostly zero values in practically all samples. Features with such low variance provide no or very limited useful information for a learning algorithm (N. Kumar and Sandeep Kumar 2021). As such, they are good candidates for removal, as they would only add computation cost without any benefit for the model's predictive capabilities.

This comprehensive and feature-wise deep dive is extremely important to better understand what kind of data is being used as the raw material for the ML models. This is crucial to be able to make more informed and better decisions in the data cleaning and transformation steps that follow.

2.14.8 Correlation Analysis

To get even more insights into the inter-feature relationships present in the dataset, the full correlation matrix was calculated, usually using Pearson correlation coefficient for each pair of numerical features, and then visualised as a heatmap for a quick and intuitive graphical representation of the linear correlations between each pair of the selected features. As expected, strong positive and negative correlations between many features were detected, which is a very common behaviour for complex network flow data. As an example, one of the most evident correlations found in many cases is between the number of Total Fwd Packets and the Fwd Header Length , as obviously sending more packets results in a larger total header length. At the same time, all different inter-arrival times often form a group of highly inter-correlated features (e.g. Flow IAT Mean and Flow IAT Max).

In addition to this exploratory purpose, finding strong correlations (positive or negative) between pairs of independent features is useful even for tree ensembles like XGBoost, which are usually more robust to such correlations than are linear models (regression, logistic regression, SVM, etc.). Features that are highly correlated with each other are usually either redundant or correlated with the target, but not with each other. They do not usually cause problems with tree ensembles, but can still add to the computational complexity of training and inference, and thus should still be identified in order to be able to remove one of the highly correlated features in the pair and thus simplify the model without a significant loss of information (Habeeb and Babu 2024). In this case, in-depth understanding of all such inter-feature correlations present in the CIC-IDS2017 dataset also provides even more knowledge about the flow data itself and its patterns, which can be effectively used by the NIDS.

2.14.9 Identified Challenges and their Implications for the Methodology

Detailed and iterative initial EDA of the CIC-IDS2017 dataset clearly revealed several critical and deeply interrelated challenges. As it was mentioned before and emphatically became apparent throughout the EDA process, these challenges are a major problem: they define and dictate all further critical and non-trivial choices that should be made in the methodology of this study, which, in the end, develops a functional NIDS based on the CIC-IDS2017 dataset.

1. Severe Class Imbalance An enormous (extreme) imbalance in the number of benign and attack flows was one of the most important aspects of the CIC-IDS2017 dataset, as it was repeatedly shown, discussed, and especially emphasised during the EDA process. This huge difference in the number of examples in the two major classes of the dataset is by far the most significant challenge that this study is faced with and has to solve. If not properly accounted for or handled, this severe class imbalance will make it *extremely likely* for the trained machine learning model to be *highly biased* towards the *greatly overrepresented benign traffic* class. Such an imbalance will most likely result in a built NIDS that would have an *alarmingly high false negative rate (missed intrusions)* for the attack classes, which makes the NIDS unreliable in real security applications, even if its overall accuracy is high (Mahfouz et al. 2022). Therefore, using at least some of the advanced imbalance handling methods that were discussed in the Section **??** becomes a requirement and a top priority when building the necessary pre-processing pipeline.

2. Missing and Infinite Values The common (very frequent) presence of NaN and Infinity values, especially in the important rate-based features like Flow Bytes/s and Flow Packets/s , is a very significant data quality problem. The reasons for these suspicious values present in the CIC-IDS2017 dataset are the specific conditions and operations on flow records in certain corner cases (that result in 0 duration or other issues) in CICFlowMeter (Imed Sharafaldin, Habibi Lashkari, and Ghorbani 2018). Improper or inconsistent treatment of these NaN and Infinity values in pre-processing can lead to errors when training the model or even when performing data analysis, can result in noisy feature distributions, and, in general, add a significant amount of noise to the data that can confuse the learning algorithm and seriously impact its learning and final performance. Therefore, a consistent and well-justified approach to these values is clearly needed to be used in this study, including their imputation (mean, median or other imputation methods) or, in extreme cases, removal of such features from the data.

3. Redundant and Highly Correlated Features In addition to a comparatively large number of 78 features in CIC-IDS2017, the EDA process clearly showed a significant amount of redundancy and high inter-feature correlations between many of these features. For example, there were numerous pairs of features that contain highly similar information and are thus *multicollinear*. This is an issue that needs to be taken care of as it, among other issues, needlessly increases the computational complexity and training time of the ML model and *can also lead to decreased performance* of the ML model by either adding noise or by giving too much weight to some aspects of the input flow data (N. Kumar and Sandeep Kumar 2021). This also makes the interpretation of feature importance scores difficult as correlated features tend to affect each other's perceived importance scores. Therefore, either an explicit feature selection process (selecting the most discriminative features to keep and removing all others) is required or an effective intrinsic method of dealing with correlated features within the model should be used, such as XGBoost's built-in ensemble method of creating decision trees that automatically alleviates the issue to a certain extent (**guha2017tutorial**).

4. Presence of Outliers Presence of extreme outliers in the data for a number of numerical features was confirmed during the feature-wise EDA. However, the nature of these extreme outliers is still unclear: they can either be true positive or true anomalies (i.e. real intrusions, which need to be detected by the NIDS) or can be simply true benign flows, which are very rare but still possible, or finally could even be an artefact of data collection or preparation that needs to be removed from the data. All these options are reasonable and very hard to tell apart, as there is currently no direct way to look at the data and know which of the flows are indeed outliers and which ones are not. The choice of how to treat these outliers (remove them, use some transformations like a logarithm to make them less extreme or leave them alone) thus becomes a crucial and difficult question as each option will have a direct impact on the trained model's behaviour and, most importantly, on the number of false positives that it will generate. Therefore, a well-thought and clearly justified treatment of these suspicious outliers is necessary to avoid removing the true positive outliers but still handle these extreme data points in a way that they do not negatively impact the learning algorithm (Al-Qerem, Almomani, and Al-Khateeb 2022).

5. Categorical Feature Encoding Finally, there were one or more categorical features, like Protocol , which need a particular way of being encoded so that the learning algorithm can easily use them and can also be sure that they do not contain any artificial relationships, such as an implicit ordering of categories for a nominal categorical feature. For example, if a simple integer mapping would be used to encode the categorical Protocol feature, the learning algorithm would probably learn and apply an ordering between these protocols, which is not present in reality and will lead to a significant drop in performance. Therefore, an appropriate encoding for nominal categorical features (like One-Hot Encoding) and separate encoding for the target Label column (like Label Encoding) will be required.

2.14.10 Addressing EDA Insights in the Methodology

Insights from the very detailed and time-consuming initial dataset exploration have most definitely and to a great extent defined the rest of the study's methodology. In other words, the entire study, the methodology, which builds the proposed XGBoost-based NIDS using CIC-IDS2017, directly addresses and solves all the challenges of this dataset that were revealed during EDA.

- Imbalance Handling Methods: Since a class imbalance is a very serious and severe issue for this dataset, as repeatedly shown and concluded during EDA, the methodology clearly and most importantly includes techniques that effectively address it. This means a very specific and careful use of oversampling for attack classes on the training dataset (SMOTE or ADASYN for minority classes, but strictly in a way that does not cause data leakage) or, alternatively, very precise and controlled use of cost-sensitive learning through specific settings and the XGBoost algorithm itself (adjusting scale_pos_weight specifically for attack classes). These approaches and the required data pre-processing are, by no means, an afterthought but are, rather, absolutely critical for balanced and successful detection of all attack classes, especially the minority ones (Mahfouz et al. 2022).
- Treatment of Missing and Infinite Values: In order to remove a problem of NaN and Infinity values in the data, especially in the important rate-based features like Flow Bytes/s and Flow Packets/s , which are a common data quality issue, a very precise and robust approach to these values has to be used in this study. The first necessary step is to convert all Infinity values to NaN to have a consistent format for all missing data, and then a mean imputation method can be used to fill all NaN s with the corresponding feature's mean value. This approach was chosen because it is computationally fast and efficient and also will not alter the distribution of the values for the affected features. The effect of this imputation on the variance of features and potential bias would also be closely monitored.
- Categorical Encoding: The use of an appropriate and precise categorical encoding for all nominal categorical features (One-Hot Encoding) and specifically separate encoding for the target Label column using Label Encoding ensures that the learning algorithm both can easily interpret the input data and is also not biased by some artificial ordinal or categorical relationships.
- Feature Scaling: Even though tree-based models like XGBoost are robust to feature scaling, all numerical features will be standardised using StandardScaler to have a mean of 0 and standard deviation of 1. This is done to help the model with convergence when training, to make regularization more effective, and to prevent the features with larger numerical ranges from having an undue impact on the learning process.
- Balanced Evaluation Metrics: The per-class evaluation metrics are heavily emphasised and carefully used because of the severe class imbalance in CIC-IDS2017 dataset. Due to this data issue, using simple metrics, such as overall accuracy, is clearly not enough and can be very misleading. Therefore, this study will focus more on a comprehensive set of per-class metrics , specifically detailed Precision, Recall and F1-score for *each individual attack type* as well as the 'Benign' class, and even FPR and a granular, informative Confusion Matrix . This will provide a much more reliable and actionable result of the NIDS's performance on all classes of network traffic (Al-Qerem, Almomani, and Al-Khateeb 2022).
- Tuning-Informed Hyperparameter Selection: The choice of hyperparameters to tune for the XGBoost model in this study is significantly influenced by the findings from the EDA. Special

attention will be given to hyperparameters that directly impact regularization (such as gamma , reg_alpha , reg_lambda) and, most importantly, class weighting (such as scale_pos_weight). Tuning these parameters carefully is critical to address overfitting and enhance the model's generalisation, and also to specifically handle the severe class imbalance.

- Baseline for Comparative Analysis: A very comprehensive and detailed understanding of CIC-IDS2017 dataset, that was achieved in this very EDA section, provides a robust baseline for a later comparison of the optimised XGBoost model against other carefully selected machine learning algorithms. This will guarantee that all comparative models are trained and then evaluated on exactly the same, well-understood and appropriately pre-processed data, and thus ensure a fair and robust comparison (Shafi and M. Rahman 2022).

The initial extensive exploration of the CIC-IDS2017 dataset is, by no means, just a formality in this study but is, instead, an essential and crucial starting point that guarantees that all further steps in the pipeline of this machine learning study are not only technically sound and scientifically rigorous but also strategically and intimately connected to the intrinsic and complex challenges and characteristics of the selected dataset. This carefully built groundwork in turn greatly improves the overall validity, reliability, and real-world applicability of this work's results.

2.14.11 Conclusion of Dataset Exploration

The initial detailed exploration of the CIC-IDS2017 dataset has most certainly and beyond reasonable doubt proven itself an immensely valuable and even an essential first step of the entire machine learning pipeline for this work. This process of data inspection and careful analysis has not only provided a wealth of critical information about the realistic composition of this dataset, including its inherent attack diversity, but also has revealed several profound and persistent challenges, including a severe class imbalance, very frequent missing and infinite values, and even a significant amount of inter-feature correlations. These important findings have then been leveraged not only to provide an actionable and deep understanding of the data's fundamental properties but also have directly informed and, most importantly, strongly justified all further decisions throughout this study's methodology, from an exact choice of necessary data pre-processing techniques all the way to a specific set of evaluation metrics. This careful and extensive preliminary analysis, in turn, forms a robust and very essential foundation that truly guarantees that the entire development and then very rigorous evaluation of the proposed XGBoost-based NIDS is firmly grounded in a very nuanced and comprehensive understanding of the dataset's unique challenges and its actual intricacies.

2.15 Deep Dive into Ensemble Learning

2.15.1 Introduction

As we explore in this deep dive into ensemble learning, no single machine learning algorithm can be crowned as the definitive best for every possible task and dataset. Each model has its inherent advantages and weaknesses, and the choice of the "best" solution often depends on the specific context of the problem. This unpredictability and variability in model performance has led to the development and widespread adoption of ensemble learning (Al-Garadi, Ali Mohamed, et al. 2020). Ensemble methods take a unique approach by combining the predictions from multiple "weak learners" to create a more powerful and accurate "strong learner." The intuition behind this approach is simple: a decision that reflects the consensus of a group of experts is usually more robust and precise than that of any individual expert, however skilled. This is especially true in complex, multi-dimensional domains like Network Intrusion Detection Systems (NIDS), where malicious patterns can be subtle and sophisticated (Aldhubaib and N. Ali 2024).

The ensemble strategy is backed by a strong theoretical foundation in statistics and computation. At a high level, all ensemble methods aim to reduce bias , variance , and overfitting , common issues that individual models often face. They achieve this by combining diverse models to "average out" errors, diminish the effect of noisy data, and improve the model's ability to generalize to new, unseen data. This is based on the principle that if models make different types of errors, an ensemble can reduce the overall error rate. In this chapter, we will delve into the world of ensemble learning, discussing its underlying principles, the main types of ensemble methods (bagging, boosting, and stacking), and the workings of key algorithms in each category. We will also focus on their practical application and effectiveness in the challenging domain of NIDS, drawing on recent research to highlight their importance in improving cybersecurity measures.

2.15.2 The Basics: Foundations of Ensemble Learning

The primary principle driving ensemble learning is the concept of diversity . For an ensemble to outperform its individual learners, these learners must make different (or diverse) predictions. If all base models make the same errors, combining them will not yield a better result. This diversity can be introduced in various ways:

- Data Manipulation: Training base models on different subsets of the training data (e.g. bootstrapping in bagging).
- Feature Manipulation: Training base models on different subsets of features (e.g. random feature selection in Random Forests).
- Algorithm Diversity: Using different types of learning algorithms as base models.
- Parameter Diversity: Using the same algorithm but with different hyperparameters.

The aggregation strategy then dictates how these diverse base model predictions are combined. This could be simple voting (for classification), averaging (for regression), or more complex meta-learning approaches. The power of an ensemble lies in leveraging these diverse perspectives.

2.15.3 Bias-Variance Trade-off

One of the key concepts that explains the power of ensemble methods is the bias-variance trade-off . In the context of supervised learning, the total error of a model can be decomposed into three main components: bias, variance, and irreducible error.

- Bias refers to the error introduced by approximating a real-world problem, which may be complicated, by a simplified model. High-bias models (e.g. linear regression on non-linear data) tend to "underfit" the training data.
- Variance refers to the model's sensitivity to small fluctuations in the training data. High-variance models (e.g. deep decision trees) tend to "overfit" the training data, making them perform well on seen data but poorly on unseen data.

Ensemble methods are cleverly designed to address both these issues. Methods like bagging primarily reduce variance by averaging the predictions of multiple high-variance, low-bias models. On the other hand, boosting primarily reduces bias by building models sequentially, with each new model focusing on correcting the bias (errors) of the previous models, often using low-variance, high-bias weak learners. This interplay allows ensembles to achieve a better balance in the bias-variance trade-off, leading to overall better performance (Zou, Lo, and H. Kim 2021).

2.15.4 Bagging (Bootstrap Aggregating)

Bagging , or Bootstrap Aggregating, is one of the earliest and simplest ensemble methods, introduced by Leo Breiman in 1996. It is primarily used to reduce the variance of a prediction by generating additional data for training from the original dataset via combinations with replacement (bootstrapping). The base idea is to train several base models independently, and then combine their predictions, usually by voting for classification or averaging for regression.

2.15.5 Bootstrapping

Bootstrapping is a resampling technique where subsets of the original dataset are created by drawing samples *with replacement*. If the original dataset has N samples, a bootstrap sample will also have N samples but some of the original samples will appear multiple times, and some will not be chosen at all. This process is repeated M times to get M different bootstrap samples, each of which is used to train a separate base learner (Agrawal, Jain, and Rakesh Gupta 2021). Since each base learner is trained on a slightly different subset of the data, they will likely learn different patterns and make different errors, which is what provides the necessary diversity to the ensemble.

2.15.6 Workings of Bagging

Bagging can be summarized in a few simple steps:

1. Draw M bootstrap samples from the original training dataset.
2. Train M independent base learners (typically of the same type, e.g. Decision Trees) on each of the M bootstrap samples.
3. For a new, unseen data point, each of the M base learners makes a prediction.
4. Aggregate the predictions: for classification, this is usually a majority vote; for regression, it's the average of the predictions.

By averaging (or voting) over the predictions of these diverse base learners, bagging significantly reduces the overall variance of the model, without increasing bias. It works best with complex, high-variance base models like deep Decision Trees.

2.15.7 Random Forest (RF)

Random Forest (RF) is one of the most well-known and widely used bagging algorithms. It was developed by Leo Breiman in 2001 and is, in essence, an enhancement of the basic bagging approach. RF introduces an additional source of randomness into the bagging process, specifically in the tree-building process. This further decorrelates the individual Decision Trees within the ensemble and makes the RF even more resistant to overfitting than standard bagging.

A Closer Look In Random Forest, each tree is built not only from a bootstrap sample of the data but also by considering only a random subset of features at each split point within the tree (M. Al-Mutairi et al. 2025).

1. For each of the M trees:
   - A bootstrap sample of the training data is drawn.
   - At each node of the tree, instead of considering all available features for the best split, only a random subset of k features is considered. The optimal choice of k is typically $\sqrt{p}$ for classification (where p is the total number of features) and $p/3$ for regression.
   - The Decision Tree is grown to its maximum depth without pruning (or with very minimal pruning), as the ensemble's averaging effect will take care of any overfitting.
2. For prediction, the output is simply aggregated: majority vote for classification, average for regression.

This two-level randomness (data sampling and feature sampling) ensures that the individual trees are diverse and do not overfit to specific features, which results in an extremely powerful and generalizable model.

RF in the NIDS: Strengths Random Forest's advantages make it particularly well-suited for NIDS:

- Reduction in Overfitting: The ensemble nature and two-level randomness of RF provide excellent immunity against overfitting, which is crucial when dealing with noisy and high-dimensional network traffic data (M. Al-Mutairi et al. 2025).
- High Predictive Accuracy: RF consistently achieves high predictive accuracy across a wide range of NIDS tasks, from binary (attack/benign) to multi-class classification of different attack types (Shafi and M. Rahman 2022).
- Feature Importance Insights: RF inherently provides a measure of feature importance (e.g. Gini importance or permutation importance), which helps to identify the most discriminative network flow features for intrusion detection, which is a valuable insight for security analysts (L. Wang, Hong Zhang, and Q. Li 2023).

- Handles High-Dimensionality Well: RF can perform well even when the number of features is large, as it only considers a subset of them at each split.
- Robustness to Noise and Outliers: The aggregation of many trees makes it less sensitive to individual noisy data points or outliers than single models would be.

Limitations RF, despite its many strengths, has its own set of limitations:

- Less Interpretable: While individual Decision Trees are highly interpretable, an ensemble of hundreds or thousands of trees becomes a "black box", making it difficult to explain the reasoning behind specific intrusion alerts.
- Computational Cost: Training many trees can be computationally expensive and time-consuming, particularly for very large datasets and deep trees.
- Bias towards Categorical Features: Can be biased towards features with more categories when computing feature importance.

Applications and Impact in NIDS RF has been used extensively in NIDS research and has shown impressive results. For example, in (Shafi and M. Rahman 2022), RF's robust performance across various NIDS datasets is highlighted. In (L. Wang, Hong Zhang, and Q. Li 2023), RF's efficacy in classifying network intrusions is demonstrated, often outperforming traditional machine learning algorithms due to its ability to handle complex feature interactions and data noise. More recent studies on the CIC-IDS2017 dataset have continued to find RF as a strong contender for accurate intrusion detection, especially when dealing with multi-class classification problems with diverse attack types (M. Al-Mutairi et al. 2025). It has also been used for feature selection by leveraging its built-in feature importance, which can help build more lightweight and efficient NIDS models.

2.15.8 Boosting

Boosting is the opposite of bagging's parallel, independent training of multiple models. In boosting, models are built sequentially, with each new model focusing on correcting the errors made by the previous ones. The main idea is to iteratively train "weak learners" (models that are only slightly better than random guessing) and combine them into a strong learner. The key is that each new weak learner is trained to give more attention to the data points that the previous models misclassified or handled poorly, thus sequentially improving the overall model's performance. The primary goal of boosting is to reduce bias and transform many weak, simple models into a single, highly accurate one (Friedman 2001).

2.15.9 The General Principle of Gradient Boosting Machines (GBM)

Most modern boosting algorithms are built on the core concept of Gradient Boosting Machines (GBM) (T. Chen and Guestrin 2016). The fundamental idea behind GBM is to combine weak learners (most commonly, Decision Trees, specifically "regression trees") by fitting each new tree to the residuals (errors) of the previous step. Rather than fitting the new tree directly to the original target variable, the new tree is fit to the negative gradient of the loss function with respect to the current model's predictions. This iterative process allows the model to progressively reduce the loss function, building an ensemble that specifically targets the remaining errors. GBM has several key characteristics:

- Sequential Learning: Trees are added one at a time, and the previous trees remain unchanged.
- Loss Function Optimization: The process is driven by minimizing a differentiable loss function (e.g. mean squared error for regression, logarithmic loss for classification).
- Weak Learners: Simple models, usually shallow Decision Trees (stumps or trees with few splits), are used as base learners.
- Shrinkage (Learning Rate): Each new tree's contribution to the ensemble is scaled by a small learning rate, which helps prevent overfitting and improve generalization.

From this general framework, several highly optimized and popular boosting algorithms have been developed, revolutionizing machine learning.

2.15.10 Extreme Gradient Boosting (XGBoost)

XGBoost (eXtreme Gradient Boosting) is a highly optimized, scalable, and powerful implementation of the gradient boosting framework. Created by Tianqi Chen, it has gained massive popularity and often achieves state-of-the-art performance in a wide range of machine learning competitions and real-world applications since its release (T. Chen and Guestrin 2016). XGBoost builds on the core ideas of GBM and introduces several important enhancements that significantly improve its efficiency, scalability, and predictive power.

Inside the Black Box XGBoost's remarkable performance is due to a combination of algorithmic and system-level optimizations:

1. Regularization: Unlike traditional GBM, XGBoost includes explicit L1 (Lasso) and L2 (Ridge) regularization terms in its objective function. These regularization terms penalize the complexity of the model (e.g. the number of leaves and the magnitude of leaf weights), which actively prevents overfitting and encourages model generalization. This is a key feature, especially for noisy data like network traffic (Habeeb and Babu 2024).
2. Advanced Tree Construction: XGBoost uses a 'level-wise' or 'depth-wise' tree growth strategy, also known as Exact Greedy Algorithm or Approximate Greedy Algorithm for large datasets. It also uses a smart split-finding algorithm that includes sparsity-aware split finding, which can efficiently deal with missing values and zeroes by learning the best direction for missing values.
3. Shrinkage and Column Subsampling: Like GBM, XGBoost also uses a learning_rate (or eta) to shrink the contribution of each individual tree, preventing any one tree from dominating the overall prediction and making the learning more conservative. It also uses column subsampling (like Random Forest's feature bagging) to randomly select a subset of features for each tree, which further reduces variance and speeds up computation (M. Al-Mutairi et al. 2025).

4. Tree Pruning: XGBoost uses a 'max_depth' parameter and a gamma parameter. gamma specifies the minimum loss reduction required to make a further partition on a leaf node. If a split does not meet this threshold, it is pruned. This allows for much more effective pruning than the traditional greedy algorithm, which only prunes after the tree is fully grown.
5. Handling Missing Values: XGBoost can automatically learn the best way to handle missing values by having default directions for splits when a value is missing, which is highly useful for real-world datasets where missingness is a common occurrence.
6. Parallel Computing: At a system level, XGBoost is designed to support parallel processing of tree construction. While trees are added sequentially, the inner loops (e.g. finding the best split for a node) can be parallelized across CPU cores, which can greatly speed up training.

Key Advantages

- High Accuracy: Consistently provides state-of-the-art predictive performance across a wide range of classification and regression tasks.
- Robustness against Overfitting: The built-in regularization, shrinkage, and column subsampling mechanisms offer strong protection against overfitting.
- Scalability and Efficiency: Designed for parallel computation and efficient memory usage, making it highly effective for large datasets.
- Handles Missing Data Automatically: Its sparsity-aware split finding can intelligently handle missing values without requiring explicit imputation beforehand (although pre-imputation can still be helpful).
- Flexibility: Supports custom objective functions and evaluation metrics, allowing it to be tailored to specific problem needs.
- Feature Importance: Provides robust feature importance scores, which can help in understanding the factors that drive predictions.

2.15.11 Computational Complexity and Resource Requirements

Ensemble methods, especially those involving large numbers of deep trees like Random Forests or boosting algorithms like XGBoost, are computationally intensive. They require significant CPU and memory resources for both training and inference (Aldhubaib and N. Ali 2024). This can be particularly challenging in real-time or near-real-time NIDS, where decisions need to be made swiftly as packets arrive. Deploying these models in operational environments may necessitate high-performance computing infrastructure or optimisations that can scale with the massive, streaming nature of network data.

2.15.12 Model Complexity and Interpretability

The increased complexity that comes with ensembles can lead to models that are difficult to interpret. Understanding the basis for an ensemble's decision can be more challenging than with simpler models.

In the context of NIDS, where explaining or auditing decisions may be necessary for network forensics or compliance reasons, this lack of interpretability can be a significant drawback. Techniques to improve interpretability, like feature importance rankings or model-agnostic interpretation methods, may need to be applied, but these can add overhead and may not fully resolve the issue (Y. Liu and S. Li 2021).

2.15.13 Integration with Existing Infrastructure

Deploying ensemble-based NIDS into existing network security architectures can be non-trivial. It may require changes to data handling processes, updates to integration with other security tools (like SIEM systems), or adaptations in network hardware to support the required computational load. Careful consideration must be given to how these systems interface with current infrastructure, including considerations around data privacy and security, especially when deploying to environments with stringent regulatory requirements (Albanese, Al-Shaer, and Pezzella 2021).

2.15.14 Overfitting with Improper Tuning

While ensemble methods are less prone to overfitting, they are not immune, particularly if improperly configured. Using too many models in an ensemble, especially deep or complex ones, can lead to overfitting to the noise in the training data rather than underlying attack patterns. In NIDS, where the availability of labelled attack data can be limited, there's a risk of building a system that performs well on known threats but poorly on novel or emerging ones. Regularisation, careful tuning of ensemble size, and validation against diverse and representative datasets are essential to mitigate this (Gautam and Raj 2021).

2.15.15 Maintaining Performance with Evolving Threats

The effectiveness of an NIDS can degrade over time as attackers adapt and develop new techniques. While ensemble methods are generally good at generalisation, continuously evolving the ensemble model to keep pace with new types of network traffic and attack strategies can be challenging. This might involve regular retraining, updating the model with new data, or dynamically adjusting ensemble composition. Such maintenance requires ongoing effort and expertise to ensure the NIDS remains effective (M. A. Khan and A. Rahman 2021).

2.15.16 Dataset Specificity and Transferability

Ensemble models can sometimes be very specific to the datasets on which they were trained. Transferring a model from one network environment to another, or from one type of network traffic to another, can lead to a drop in performance if the underlying data distributions differ significantly. This issue of transferability is particularly acute in NIDS, where the operational environment and traffic patterns can vary widely from the conditions seen in training datasets (Wang and J. Chen 2023).

2.15.17 Hyperparameter Tuning and Configuration

Ensemble learning algorithms come with a range of hyperparameters that need to be tuned for optimal performance. This tuning process can be complex and computationally expensive, particularly for large ensembles or those using sophisticated base learners. In NIDS, where the cost of false negatives (missed intrusions) can be high, finding the right configuration that balances sensitivity, specificity, and computational efficiency is crucial but non-trivial (Z. Li and Y. Zhang 2022).

2.15.18 Increased Computational Cost and Training Time

The most direct drawback of ensemble methods is that they generally have a higher computational cost and require longer training times compared to using single models. Bagging and Random Forest involve training several independent models, while boosting trains models sequentially. Stacking, with its multi-layered architecture and cross-validation for meta-feature generation, is typically the most computationally expensive (Aldhubaib and N. Ali 2024). This can be a significant challenge, especially with very large NIDS datasets or in environments where models must be deployed or retrained frequently.

2.15.19 Reduced Interpretability (Black-Box Nature)

While Decision Trees are known for their interpretability, the ensemble of many trees (Random Forest, XGBoost, LightGBM, CatBoost) or the combination of different model types (stacking) leads to a more "black-box" approach. It becomes significantly harder to interpret exactly *why* a decision was made or *which specific features* were responsible for a given intrusion alert (Kasongo and Y. Sun 2021). This reduced transparency can affect trust and adoption among security analysts, hamper troubleshooting and post-incident analysis, and also complicate efforts to reverse-engineer attack techniques, pointing to an important research direction of Explainable AI (XAI) for NIDS.

2.15.20 Hyperparameter Complexity

Ensemble methods, particularly gradient boosting algorithms like XGBoost, LightGBM, and CatBoost, involve numerous hyperparameters that can greatly affect performance. Finding the right balance for these parameters (learning rate, tree depth, regularization terms, subsampling rate, etc.) is critical but can be a complex and time-consuming process, often involving exhaustive grid search, random search, or more sophisticated Bayesian optimization. Suboptimal hyperparameter settings can lead to underfitting or, more problematically, overfitting, thus degrading the performance of the NIDS (Habeeb and Babu 2024).

2.15.21 Vulnerability to Adversarial Attacks

Like other machine learning models, ensemble methods, including deep learning ensembles, can be vulnerable to adversarial attacks. Adversaries can craft subtle, often imperceptible perturbations to network traffic (adversarial examples) that lead the ensemble NIDS to misclassify malicious traffic as benign (evasion attacks) or as a different type of attack. The added complexity of ensemble models can sometimes make them more difficult to defend against certain adversarial manipulations, as vulnerabilities might be distributed across the different base learners (J. Kim, Hwang, and Jo 2021).

2.16 Conclusion

Ensemble learning is a major shift in the way we approach machine learning problems. Instead of relying on the strengths of a single model, we can combine multiple models to compensate for individual weaknesses, leading to more accurate, stable, and generalizable machine learning systems. Bagging methods like Random Forest work by averaging out the predictions of independently trained models on bootstrapped datasets, thus reducing variance. Boosting algorithms, such as the highly optimized XGBoost, LightGBM, and CatBoost, build models sequentially to focus on and correct the errors of previous learners, effectively reducing bias and achieving high performance. Advanced methods like stacking can further improve performance by using a meta-learner to optimally combine the predictions of diverse base models.

Ensemble learning offers several significant benefits in the context of Network Intrusion Detection Systems. It helps improve the accuracy of detection by harnessing the combined power of multiple models, it enhances robustness by reducing the likelihood of being evaded by complex attack patterns, and it addresses class imbalance by effectively leveraging information from both minority and majority classes. Feature importance analysis offered by ensembles provides valuable insights into the nature of network traffic and intrusions. Ensembles also help in improving the generalization ability of NIDS to unseen and novel threats. However, challenges such as increased computational requirements, reduced interpretability, hyperparameter tuning complexity, and vulnerability to adversarial attacks remain. Nevertheless, ongoing research and development of techniques like Explainable AI (XAI) and real-time hyperparameter optimization are actively addressing these issues and will continue to refine and reinforce the role of ensemble learning in building advanced, resilient, and adaptive NIDS for our increasingly interconnected world.

2.17 Deep Dive into XGBoost: Principles, Advantages, and Hyperparameters

2.17.1 Introduction to XGBoost

In today's fast-evolving machine learning landscape, where the quest for highly accurate, scalable, and efficient predictive models is unending, one algorithm stands out as a stalwart: XGBoost, which stands for eXtreme Gradient Boosting. Since its introduction by Tianqi Chen in 2016 (T. Chen and Guestrin 2016), it has not only been a winner in numerous data science competitions but also found its way into production in a wide variety of real-world applications, ranging from fraud detection and recommendation systems to, most relevant to this dissertation, Network Intrusion Detection Systems (NIDS) (M. Al-Mutairi et al. 2025; Habeeb and Babu 2024). The reasons for its widespread popularity are not arbitrary; they are the fruits of a sophisticated algorithm design that carefully balances predictive performance, computational efficiency, and scalability.

XGBoost is an implementation of the broader family of Gradient Boosting Machines (GBM) (Al-Garadi, Ali Mohamed, et al. 2020). However, what truly distinguishes it and earns it the moniker "eXtreme" are the numerous algorithmic and system-level optimizations that dramatically improve its performance, robustness, and speed. These improvements help it scale to large datasets, handle missing values, and reduce the dreaded overfitting, all the while maintaining state-of-the-art accuracy. For the challenging, high-stakes problem of NIDS, where enormous volumes of noisy, high-dimensional, and often highly imbalanced network traffic data must be sifted through to detect rare and sophisticated cyber threats, XGBoost provides a compelling, highly effective solution (Aldhubaib and N. Ali 2024). This chapter is an in-depth study of XGBoost, where its core principles are explained, unique advantages are highlighted, and most importantly, the effects of key hyperparameters are described in detail.

2.17.2 Core Principles of XGBoost

In order to understand XGBoost, one must first be familiar with a few key concepts: ensemble learning, gradient boosting, and the choice of the base learner (weak learner). These concepts form the backbone of this powerful learning algorithm.

2.17.3 Ensemble Learning and Boosting

XGBoost is based on the principle of ensemble learning, a machine learning paradigm that combines the predictive power of multiple "weak learners" to create a more accurate and robust "strong learner" (Zou, Lo, and H. Kim 2021). This "wisdom of the crowd" often leads to superior predictive performance than any of the individual models could achieve on their own. XGBoost is a part of a specific class of ensemble methods known as boosting algorithms.

Unlike bagging methods (like Random Forest, for example), which train several models independently of one another and then average their outputs to create the ensemble, boosting algorithms build their ensemble in a sequential, additive manner. Each new weak learner in a boosting ensemble is trained to improve upon the performance of the existing ensemble of learners. This sequential refinement of the ensemble allows the model to focus more and more on the "difficult" data points that the current ensemble finds hard to predict, iteratively reducing the ensemble's overall error (Agrawal, Jain, and Rakesh Gupta 2021). The end goal of boosting is to reduce bias: take many weak, simple (and hence low-variance and high-bias) models and transform them into one single, more complex and powerful model.

2.17.4 Gradient Boosting Machines (GBM)

The immediate precursor to XGBoost is a machine learning algorithm known as the Gradient Boosting Machine (GBM). The idea behind it is to frame the problem of ensemble learning as an optimisation problem of minimising some differentiable loss function (T. Chen and Guestrin 2016). In other words, instead of training a new weak learner on the residuals (or the difference between the actual target value and the prediction from the previous iteration of the model), GBM fits each new tree to the negative gradient of the loss function with respect to the current ensemble's predictions.

Suppose the current ensemble model at step m is $F_m(x)$ and the next new weak learner to be added to it is $h_m(x)$, then the updated model is given by $F_{m+1}(x) = F_m(x) + \gamma_m h_m(x)$, where γ_m is a step size. The new weak learner $h_m(x)$ is then trained to predict the "pseudo-residuals," which is proportional to the negative gradient of the loss function $L(y, F_m(x))$ with respect to $F_m(x)$:

$$r_i = -\left[\frac{\partial L(y_i, F_m(x_i))}{\partial F_m(x_i)}\right]_{F_m(x)}$$

This ensures that the new tree is explicitly pushing the ensemble model towards a more optimal solution, and this process is repeated many times, thus minimising the overall loss function. The continuous refinement of the model is directly aimed at the parts where the model is performing poorly, resulting in large reductions in the prediction error.

2.17.5 Decision Trees as Weak Learners

In XGBoost and most GBM implementations, the weak learners are almost exclusively Decision Trees, and more specifically, "regression trees". These are typically shallow trees, often called 'stumps' or trees of some small, pre-specified maximum depth. The choice of decision trees as the weak learner is deliberate for several reasons:

- Non-linearity: Trees can model complex non-linear relationships and interactions between features, which is essential for modelling complex datasets like network traffic.
- Interpretability (Individual Tree): While the entire ensemble can be a black-box model, individual shallow trees are relatively easy to interpret.
- Robustness: They are robust to feature scaling and can handle mixed types of data (numerical and categorical features).
- Efficiency : Training shallow trees is computationally efficient.

Each new tree is optimised to learn the errors in the predictions of the previous ensemble and its contribution is often "shrunk" by a learning rate to ensure a conservative, gradual improvement and to also prevent overfitting (Zou, Lo, and H. Kim 2021).

2.17.6 XGBoost: Algorithmic Enhancements and Mechanics

It is the algorithmic and system-level improvements to the basic GBM architecture that make XGBoost special. XGBoost is the result of many such improvements that were carefully designed and optimised to provide top-notch performance, speed, and scalability.

2.17.7 Regularization in the Objective Function

One unique feature of XGBoost that sets it apart from most GBM implementations is the inclusion of explicit regularization terms in the objective function itself. While most previous GBM algorithms would resort to tree pruning for regularisation, XGBoost's objective function for the t^{th} iteration is given by:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Where l is the differentiable loss function (usually squared error loss or cross-entropy loss), $f_t(x_i)$ is the new tree being added, and $\Omega(f_t)$ is the regularisation term for the new tree. This regularisation term typically consists of L1 and L2 penalties:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Where T is the number of leaves in the tree, w_j is the weight (prediction) of leaf j, γ is the L1 regularisation term, and λ is the L2 regularisation term.

- L1 Regularization (α or reg_alpha): The L1 penalty term biases the optimisation algorithm to prefer trees with fewer leaves and feature interactions (simple trees with fewer splits), and can also be useful in feature selection as it pushes the weights of some feature paths towards zero.
- L2 Regularization (λ or reg_lambda): The L2 term discourages large leaf weights, and in a way smooths out the model by reducing its reliance on individual instances, helping the model to generalise better and thus combat overfitting (Habeeb and Babu 2024).

By building these explicit penalties for model complexity into the objective function itself, XGBoost naturally builds simpler, more generalisable trees, which is an extremely powerful way of guarding against overfitting, especially for noisy, high-dimensional datasets such as network traffic.

2.17.8 Sparsity-Aware Split Finding

Datasets, especially in areas such as NIDS, are often likely to contain sparse features or a large number of missing values (N. Kumar and Sandeep Kumar 2021). XGBoost is specifically designed to deal with these efficiently via a sparsity-aware split-finding algorithm. Instead of requiring explicit imputation of missing values prior to training (though this may still be required for some types of features before training), XGBoost learns the best direction for missing values at each split. For every feature and for each potential split, the algorithm learns two default directions for missing values: one that sends all missing values in the left child, and the other that sends them to the right child. The optimal default direction (minimising the loss) is then selected, thus learning to handle missing values for each feature (T. Chen and Guestrin 2016). This not only simplifies the data preprocessing pipeline, but can often lead to more robust models as well as the algorithm has explicitly accounted for the value (or lack thereof) of missing features.

2.17.9 Approximate Greedy Algorithm and Weighted Quantile Sketch

For large datasets that do not fit in memory, computing the exact best split at every single node of every tree can be infeasible. XGBoost instead uses an Approximate Greedy Algorithm (T. Chen and Guestrin 2016). Instead of searching all the possible split points to find the exact best split, this algorithm proposes a set of candidate split points. It does so by building a "weighted quantile sketch" over the distribution of the feature, and then searching for the best split among these candidate split points only. This drastically reduces the computation time while still yielding very high accuracy. The weights in the weighted quantile sketch correspond to the second-order gradients so that the algorithm is biased towards splits that are more likely to reduce the objective function.

2.17.10 Shrinkage (Learning Rate)

One extremely useful method of regularisation that is often used in gradient boosting to prevent over-fitting (and is also used in XGBoost) is called shrinkage, and is controlled by the learning_rate (or eta) hyperparameter. The effect of this hyperparameter is to shrink the contribution of each new tree added to the ensemble by a small factor equal to the learning rate. In other words, it scales down the predictions of each new tree by the value of the learning rate. This means that each successive tree has more "room" to fit the errors left by the previous ensemble of trees, resulting in a more conservative, gradual training process (Aldhubaib and N. Ali 2024). A smaller learning rate often requires a greater number of trees (n_estimators) to train but usually results in a better, more robust model with better generalisation performance. It also acts as a regularisation mechanism, as no single tree is allowed to dominate the ensemble or over-correct the errors.

2.17.11 Column Subsampling (Feature Bagging)

As in Random Forest, XGBoost also employs column subsampling (also known as feature bagging). At each boosting round (that is, at each new tree being built), only a random subset of the available features are considered for the best split. This has two effects:

- Reduces Overfitting: By introducing a random element in the choice of features, it helps prevent individual trees from over-fitting to specific features and thus improves the robustness of the ensemble.
- Speeds up Computation: Evaluating fewer features at each split greatly speeds up the tree-building process.

XGBoost has various options for column subsampling: colsample_bytree (subsample features per tree), colsample_bylevel (subsample features per split level), and colsample_bynode (subsample features per node), allowing fine-grained control over this regularisation mechanism (M. Al-Mutairi et al. 2025).

2.17.12 Tree Pruning

XGBoost has a unique tree pruning strategy. Rather than greedily grow each tree to some maximum depth and then prune it back (post-pruning), XGBoost has the ability to do pre-pruning using the gamma parameter. gamma specifies the minimum loss reduction required to make a further partition on a leaf node. If a potential split does not provide a sufficient loss reduction as specified by gamma , it will simply not be made. This direct approach to pruning usually results in faster tree construction and also prevents overfitting by not allowing unnecessary complex branches (T. Chen and Guestrin 2016).

2.17.13 Parallel Computing

Although the boosting process itself is sequential in nature (trees are built one after the other), XG-Boost can still achieve significant speedups through parallel computation at the system level. The inner loop of tree construction (finding the best split for each node) is the most time-consuming part, and XGBoost parallelizes this step so that different features or different data instances can be processed in parallel across multiple CPU cores. This system-level parallelization is a primary reason it can handle very large datasets so efficiently, which makes it practical for high-volume NIDS (Agrawal, Jain, and Rakesh Gupta 2021).

2.17.14 Advantages of XGBoost

The combination of the algorithmic and system-level enhancements gives XGBoost several unique advantages that make it very powerful for computationally demanding and large-scale problems such as network intrusion detection.

2.17.15 Superior Accuracy

XGBoost has state-of-the-art predictive accuracy for a wide variety of classification and regression tasks. Its gradient boosting framework coupled with strong regularization means it can learn highly complex data patterns and interactions which lead to very precise predictions. In a NIDS context, that high accuracy is paramount since even a small improvement in the detection rate can translate to a major improvement in the overall cybersecurity posture of an organization (Aldhubaib and N. Ali 2024).

2.17.16 High Speed and Scalability

The "eXtreme" in XGBoost is, in many ways, a result of its impressive speed and scalability. Thanks to its parallelization capabilities, optimized tree-learning algorithms (approximate greedy algorithm, weighted quantile sketch, etc), and efficient memory usage, XGBoost can train on large datasets of sizes that would overwhelm most other algorithms (T. Chen and Guestrin 2016). This efficiency is of course critical for a NIDS that is expected to handle enormous volumes of network traffic in real-time or near real-time settings.

2.17.17 Robustness Against Overfitting

XGBoost is quite resistant to overfitting, a common problem for complex models that tend to memorize training data. This is in large part because of its multi-pronged regularization strategy with:

- Explicit L1 and L2 regularization in objective function
- Shrinkage (learning rate) to make the learning process more conservative
- Column subsampling to inject some randomness
- Tree pruning to cut off the growth of a tree when the improvement in loss is not significant

All this means the final model generalises well to unseen data. This is of course a key necessity for intrusion detection since it is necessary to detect novel or otherwise unknown cyber threats that were not explicitly present in the training set (Habeeb and Babu 2024).

2.17.18 Effective Handling of Missing Values

Unlike many other algorithms that require manual pre-imputation of missing values or even better that are robust to missing values, XGBoost has an inherent ability to handle them with its split-finding algorithm that is sparsity-aware. It learns the direction of missing values for each split in training and hence often comes with models that are better robust without any additional preprocessing overhead (Al-Garadi, Ali Mohamed, et al. 2020). This is no small advantage for real-world network traffic datasets which are likely to contain missing or undefined features.

2.17.19 Flexibility and Customization

XGBoost is quite flexible, allowing for custom objective functions and evaluation metrics. Researchers and engineers can, in fact, tailor the model closely to the specific requirements of the problem at hand. In the NIDS setting, this would mean one could optimize for particular metrics such as recall (to minimize false negatives or missed attacks) or even a balanced F1-score rather than just overall accuracy, which is especially useful when dealing with imbalanced classes (attacks v/s normal) (Shafi and M. Rahman 2022).

2.17.20 Built-in Feature Importance

Built into XGBoost is a reliable way to obtain the importance of each feature in making predictions. The feature importance scores can be quite invaluable for gaining insight into the underlying patterns behind intrusions, like what are the most important network flow attributes that help in distinguishing malicious activity from normal traffic. This can inform security teams about the most relevant aspects to monitor and even act as a form of actionable intelligence for network administrators and security analysts (M. Al-Mutairi et al. 2025).

2.17.21 Cross-Platform Compatibility

XGBoost is available in many different programming languages (Python, R, Java, Scala, C++). This makes it accessible to a larger community of developers and researchers and also helps with its integration into a variety of existing security systems and frameworks.

2.17.22 Key Hyperparameters and Their Impact

Effectively tuning XGBoost to achieve high performance for a specific task, like a NIDS, depends greatly on understanding the various hyperparameters that the model exposes to the user and tweaking them to match the characteristics of the dataset at hand. XGBoost's hyperparameters control almost all aspects of the learning process, from the depth of the trees and regularization amounts to the learning rate and data subsampling. Choosing the right set of hyperparameters is very important; setting them inappropriately can lead to a model that underfits or overfits or is just very inefficient during training.

In a nutshell, XGBoost hyperparameters fall into 3 main categories: General Parameters, Booster Parameters, and Learning Task Parameters.

2.17.23 General Parameters

These parameters govern the general working of XGBoost.

- booster : Specifies the booster to be used. Can be gbtree (tree-based), gblinear (linear model), or dart (Dropouts meet Multiple Additive Regression Trees). The default value for this hyperparameter is gbtree , which is almost always the recommended setting due to its much higher accuracy and the non-linear relationships that tree-based models are able to capture. Note that the regularisation parameters apply only to gbtree , so setting this to something else would likely lead to overfitting.
- nthread (or n_jobs in the scikit-learn wrapper): Number of parallel threads to be used when running XGBoost. Setting it to equal the number of CPU cores available will typically result in a major speedup in training. The default value for this hyperparameter is to use as many CPU cores as are available.

2.17.24 Booster Parameters (Tree-Specific Parameters for gbtree)

These parameters control the tree-specific hyperparameters and are used to regularize and tune the decision trees themselves. These are the hyperparameters that are most commonly tuned in a typical XGBoost usage.

- eta (or learning_rate) :
  - Description : Step size shrinkage used to prevent overfitting. At each boosting step, this is used to scale the contribution of the current tree before it is added to the previous trees. In other words, after a new tree is fit, its feature weights are multiplied directly by eta .

  - Impact: This smaller eta values lead to a more conservative boosting process which needs more boosting iterations n_estimators but often leads to more robust models. Higher eta values allow the model to learn faster, but the risk of overfitting is higher.
  - Tuning Strategy: This is almost always set to a small value such as 0.01, 0.05, 0.1, 0.2, 0.3, etc, and is tuned in conjunction with n_estimators .

- max_depth :
  - Description : Maximum depth of a tree. It controls the complexity of the model by constraining the amount of room each tree has to grow.
  - Impact: More depth allows each tree to capture more complex feature interactions and non-linearities but, unsurprisingly, at the cost of a much higher risk of overfitting. Shallower trees lead to simpler models that are far more robust.
  - Tuning Strategy: This is one of the most important hyperparameters to tune in order to control overfitting. It is very commonly tuned within a range of [3, 10] for most problems. With network intrusion data in particular, it is almost always a good idea to start with moderate tree depths and then, if necessary, increase cautiously from there.

- gamma (or min_split_loss) :
  - Description: Minimum loss reduction required to make a further partition on a leaf node of the tree. You can think of it as similar to the concept of information gain used in decision trees.
  - Impact: Larger values make the algorithm more conservative by pruning splits that do not lead to a sufficient decrease in loss. It is used as a kind of threshold for pruning.
  - Tuning Strategy: This is commonly tuned in ranges such as [0, 0.1, 0.2, 0.5, 1.0]. It can be an effective regularization hyperparameter to tune alongside max_depth .

- subsample :
  - Description: Fraction of training instances (rows) to be randomly sampled for each tree.
  - Impact : Setting this to less than 1.0 (e.g. 0.7, 0.8) introduces randomness into the model, decreasing its variance, which in turn prevents overfitting (this is the exact same idea as the bagging ensemble method).
  - Tuning Strategy: This is most commonly tuned within a range of [0.5, 1.0]. Lower values decrease the variance of the model, but introduce more bias.

- colsample_bytree , colsample_bylevel , colsample_bynode :
  - Description: These parameters control the subsampling of columns (features). Specifically,
    * colsample_bytree: Fraction of columns to be randomly sampled for each tree.
    * colsample_bylevel: Fraction of columns to be randomly sampled for each level (depth) in a tree.

    * colsample_bynode: Fraction of columns to be randomly sampled for each split node.
  - Impact : Same concept as subsample . Introduces randomness in the selection of the features. By making the individual trees more diverse, this reduces variance and thus helps prevent overfitting.
  - Tuning Strategy: Usually tuned in a range of [0.5, 1.0]. They can often be tuned together for a more powerful regularization effect.

- lambda (or reg_lambda) :
  - Description : L2 regularization term on weights. Note that this is not to be confused with lambda_vartype below.
  - Impact: Penalizes large leaf weights by adding their L2 squared value to the objective. This makes the model more conservative (less sensitive to the individual training instances) and helps reduce overfitting.
  - Tuning Strategy: This is typically tuned within a wide range (powers of 10 are good) such as [1, 10, 100, 1000].

- alpha (or reg_alpha) :
  - Description : L1 regularization term on weights. Note that this is not to be confused with alpha_vartype below.
  - Impact: Encourages sparsity in the leaf weights (often leads to simpler models), acting as a form of feature selection from within the tree-structure. This also helps prevent overfitting.
  - Tuning Strategy: This hyperparameter is often tuned within a range such as [0, 0.001, 0.01, 0.1].

- num_parallel_tree :
  - Description: This is only used for Random Forest mode in XGBoost. If you set it to a value > 1, XGBoost will grow multiple trees in parallel for each boosting iteration.
  - Impact: This essentially turns the XGBoost boosting algorithm into some kind of Random Forest-like ensemble.
  - Tuning Strategy : Set this to 1 for typical boosting.

2.17.25 Learning Task Parameters

These parameters set the optimization objective and evaluation metrics.

- objective :
  - Description : Specifies the loss function to be minimised.
  - Common Values for NIDS :
    * binary:logistic : Binary classification with probability output.

        - multi: softmax : Multi-class classification with predicted class label output. Requires num_class parameter.
        - multi: softprob : Multi-class classification with predicted class probabilities for each class output. Requires num_class parameter.
    - Impact: Directly dictates the optimization target for the model. Correctly setting the objective for your NIDS task (binary vs. multi-class) is critical.
- eval_metric :
    - Description : Metric for evaluating the model during validation.
    - Common Values for NIDS (especially with imbalance) :
        - logloss : Negative log-likelihood.
        - error : Binary classification error rate (1 - accuracy).
        - merror : Multi-class classification error rate.
        - auc: Area under the ROC curve (binary classification only).
        - auprc: Area under the Precision-Recall curve (great for imbalanced datasets).
        - Custom metrics (e.g., F1-score, Recall for minority classes).
    - Impact: Determines early stopping criteria and provides an interpretable target for hyperparameter tuning. For imbalanced NIDS datasets, F1-score (macro or weighted) or AUPRC are more informative than plain accuracy.
- seed (or random_state) :
    - Description : Random seed for reproducibility.
    - Impact: Guarantees that the results from multiple runs are identical if all other parameters are unchanged.
- scale_pos_weight :
    - Description: Weight balancing factor between positive and negative instances. Highly useful for severely imbalanced binary classification tasks.
    - Impact: Raises the relative weight of the minority class (positive class), compelling the model to focus more on it, thereby increasing recall for the minority class. Should be set to (sum(negative instances) / sum(positive instances)) (Habeeb and Babu 2024). For multi-class, more elaborate weighting may be required, or resorting to external resampling techniques on the training set instead.
    - Tuning Strategy: A very important parameter for NIDS as the attacks are the minority classes. Either use the formula above or tune it carefully.

2.17.26 Tuning Strategy Implications for NIDS

The tuning of XGBoost for NIDS must be conducted with a specific strategy in mind that takes into account domain-specific considerations, most notably the severe class imbalance typically present and the critical need to maximize recall of true intrusions while controlling the false positive rate.

- Prioritize Metrics: Metrics like macro-averaged F1-score, weighted F1-score or Area Under the Precision-Recall Curve (AUPRC) on the validation set should be prioritized over overall accuracy during tuning. The recall for individual attack classes is often also of critical interest and should be tracked.
- Address Imbalance Early: scale_pos_weight should be set early for binary classification. For multi-class problems, internal class weighting or external resampling techniques (e.g., SMOTE) applied to the training set are often necessary.
- Regularization for Generalization: Due to the expected noisiness of network data, heavy regularization (tuning gamma , lambda , alpha , subsample , colsample_bytree) is often beneficial to prevent overfitting to training noise and to ensure the model generalizes well to unseen attacks.
- Iterative Tuning : A common iterative tuning pipeline is:
    1. Tune n_estimators and learning_rate (begin with a moderate learning_rate and use it to find an optimal n_estimators with early stopping).
    2. Tune max_depth and min_child_weight (related to gamma).
    3. Tune subsample and colsample_bytree .
    4. Tune regularization (lambda , alpha).
    5. Re-tune learning_rate and n_estimators with the new parameters.
- Cross-Validation: Hyperparameter tuning should always be done using cross-validation (e.g., 5-fold or 10-fold) on the training set in order to get a robust performance estimate and to avoid overfitting to a single validation set split.

2.17.27 XGBoost in Network Intrusion Detection (NIDS)

XGBoost's unique combination of algorithmic excellence and practical efficiency makes it exceptionally well-suited for the demanding and critical application of Network Intrusion Detection (NIDS) (Aldhubaib and N. Ali 2024). At the core, the key strengths of XGBoost directly map to and help overcome many of the enduring challenges found in NIDS:

- Handling Massive Datasets: Network traffic data is generated in massive quantities. XGBoost's scalability and efficient use of computational resources allow NIDS to process these vast datasets in a tractable manner, a must-have feature for real-world applications (M. Al-Mutairi et al. 2025).
- Diverse and Complex Attack Detection: The nature of cyberattacks is inherently diverse and complex. XGBoost's ensemble method, built upon learning non-linear patterns with multiple decision trees, is highly capable of capturing the breadth of intrusion types, from simple port scans to more complex DoS/DDoS attacks and web exploits (Habeeb and Babu 2024).
- Class Imbalance Mitigation: Class imbalance, where benign traffic significantly outweighs attack instances, is a notorious problem in NIDS. XGBoost's scale_pos_weight parameter and support for external resampling methods (when needed) provide robust tools for rebalancing the

learning process and ensuring that the detection of critical minority attack classes is given high recall (Mahfouz et al. 2022).

- Robustness and Generalization: The strong regularization in XGBoost leads to models that are less prone to overfit to noise in the training data. This results in a robust and generalizable NIDS, capable of detecting novel or mutated variants of known attacks, a key requirement in mitigating zero-day threats (Agrawal, Jain, and Rakesh Gupta 2021).
- Actionable Insights (Feature Importance): The model's feature importance scores are not just algorithmic byproducts; they provide actionable intelligence. They help security analysts to pinpoint which network flow features contribute most significantly to detecting malicious activity, thereby offering a data-driven way to improve network monitoring, security policies, and to gain insights into the evolving landscape of attacks (Kasongo and Y. Sun 2021).

Leveraging these capabilities and systematically tuning its hyperparameters for the task at hand enables researchers and practitioners to build and deploy highly effective and resilient NIDS based on XGBoost that can make a real difference to an organization's overall cybersecurity resilience.

2.18 Identification of Research Gaps

The thorough literature review has examined the latest developments in Network Intrusion Detection Systems (NIDS), highlighting the effectiveness of Machine Learning (ML), particularly XGBoost (Aldhubaib and N. Ali 2024). We've explored its core principles, benefits like speed and accuracy, and how its hyperparameters function (T. Chen and Guestrin 2016; Habeeb and Babu 2024). The review also assessed its successful application on challenging datasets like CIC-IDS2017, along with common pre-processing and feature engineering methods (M. Al-Mutairi et al. 2025; N. Kumar and Sandeep Kumar 2021).

However, this analysis goes beyond summarising existing knowledge; it aims to pinpoint areas that need further investigation. Finding these research gaps is crucial for creating a novel and valuable contribution (Mahfouz et al. 2022). This section identifies specific areas where current research falls short, justifying the need for this dissertation and defining its potential impact on ML-based NIDS.

2.18.1 Pinpointing Specific Research Gaps for Contribution

Building on the existing literature, this section systematically identifies specific gaps where this research can make a unique and significant contribution. These areas present opportunities to improve the effectiveness of XGBoost-based NIDS.

2.18.2 Novel Pre-processing Techniques, particularly Adaptive Imputation for Multi-Class

Although standard pre-processing methods are widely used, there's a lack of research into new or adaptively combined strategies for diverse NIDS datasets like CIC-IDS2017. Current imputation methods, like using the mean or median, may not be optimal for skewed data or when missing values are meaningful. Furthermore, while techniques like 'scale_pos_weight' and SMOTE address class imbalance, the literature lacks a comprehensive analysis of multi-class-specific imbalance handling. The challenge is not just applying SMOTE but dynamically determining optimal oversampling ratios for each minority class or developing more granular cost-sensitive strategies. This research will address this gap by investigating:

- *Adaptive and Context-Aware Imputation*: Exploring imputation strategies that consider the context of the missing value.
- *Optimized Multi-Class Resampling Integration*: Comparing different sophisticated oversampling and undersampling techniques for each minority class in CIC-IDS2017 and their interaction with XGBoost's capabilities.
- *Outlier-Aware Pre-processing*: Developing pre-processing pipelines that differentiate between genuine anomalous attacks and noise to avoid removing critical attack patterns.

2.18.3 Optimized XGBoost Hyperparameter Tuning with Multi-Class Imbalance in Mind

Identification of Research Gaps The literature review chapter has already explored some of the freshest, cutting-edge developments in the domain of NIDS, briefly demonstrated the usefulness and success of ML in general, and XGBoost in particular (Aldhubaib and N. Ali 2024). We have investigated the inner workings of its base principles, advantages such as its speed and accuracy, and how its hyperparameters impact the way the ensemble algorithm functions (T. Chen and Guestrin 2016; Habeeb and Babu 2024). The literature review also encompassed a swift assessment of the successful applications on extremely challenging datasets (such as CIC-IDS2017), as well as all of the preprocessing and feature engineering methods that are most commonly used in this particular field by ML researchers (M. Al-Mutairi et al. 2025; N. Kumar and Sandeep Kumar 2021).

At this point, it is worth noting that the literature review serves not only as an aggregation and concise summary of past and present knowledge on the topic of research; but most importantly, it is meant to further enable a thorough evaluation of the state of the art in the given field by identifying the areas in need of additional study – in other words, finding the research gaps (Mahfouz et al. 2022). This section of the dissertation precisely aims to point out the particular areas in which a noticeable void has been left by other researchers, thereby justifying the existence of this body of work, and allowing it to specify its potential contribution to the fields of ML and, more specifically, ML-based NIDS.

Pinpointing Specific Research Gaps for Contribution The present section, serving as an extension to the previous one, methodically identifies the specific gaps in the literature that the present research would aim to make a novel and non-trivial contribution to, and therefore, present open opportunities for further improvement of the effectiveness of an XGBoost-based NIDS solution.

Novel Pre-processing Techniques, particularly Adaptive Imputation for Multi-Class Standard preprocessing methods have become quite routine among the related data science community, but the research has been surprisingly scarce regarding either new or adaptively combined pre-processing pipelines for multi-class (and inherently more diverse in terms of data features) NIDS datasets such as CIC-IDS2017. The prevailing imputation strategy based on a relatively simple mean or median substitution can be far from ideal in scenarios with skewed data or when the missing value itself is significant (as is often the case in ML-based NIDS). In addition, while a number of techniques such as ‘scale_pos_weight‘ and SMOTE were demonstrated to address the challenge of class imbalance, the related literature was lacking in both sophisticated, hyperparameter-free analysis of the specificities of multi-class imbalance, and an in-depth empirical exploration of how a multi-class imbalance should be addressed in terms of XGBoost-based solutions. That is, the actual challenge of applying SMOTE (as well as its alternatives) is not necessarily in its implementation, but rather in identifying the optimal upsampling/downsampling ratio for each minority class in multi-class CIC-IDS2017, or potentially developing more fine-grained cost-sensitive alternatives. This dissertation would close this gap by further exploring:

- *Adaptive and Context-Aware Imputation*: Conducting a search for alternative imputation strategies that take into account the circumstances under which the information was lost in the first place (if applicable).
- *Optimized Multi-Class Resampling Integration*: A thorough comparative empirical investigation of various sophisticated oversampling and undersampling methods for each minority class in CIC-IDS2017, and a dedicated study of the possible ways in which these methods can affect or be affected by XGBoost’s own native capabilities.
- *Outlier-Aware Pre-processing*: In the context of outlier (or invalid) handling, further development of the NIDS data pre-processing pipeline in the sense that it can distinguish between the true anomalous nature of genuinely malicious attacks vs. mere invalid or otherwise ‘rubbish’ data that is better off being discarded so as not to remove critical patterns of attack.

Optimized XGBoost Hyperparameter Tuning with Multi-Class Imbalance in Mind The body of research that focuses on tuning XGBoost hyperparameters for NIDS invariably aims for an overall improvement of the accuracy scores, while almost none of them reported a detailed breakdown of how the hyperparameters had been tuned in the context of multi-class imbalance on CIC-IDS2017. A noticeable research gap is therefore:

- *Multi-Objective Tuning for Imbalance*: Developing and applying tuning strategies that jointly optimise for multiple objectives, such as maximising the recall for the smallest minority class at the same time as minimising the overall false positive rate.
- *Dynamic ‘scale_pos_weight‘ for Multi-Class*: A study of advanced hyperparameter tuning strategies for ‘scale_pos_weight‘ that would allow it to be set more dynamically for each minority class.
- *Impact of Regularization on Imbalance*: A more thorough empirical study of the effects of the XGBoost’s regularization hyperparameters in the context of handling class imbalance and their interaction with class weighting techniques as a way to prevent overfitting.

Focus on Specific Attack Types within CIC-IDS2017 While overall detection rates can be high, XGBoost and its previous iterations are still seemingly unable to recognise specific rare attack types within CIC-IDS2017 as precisely as other attacks (examples include ‘Heartbleed‘, ‘Infiltration‘ (N.

Kumar and Sandeep Kumar 2021; Al-Qerem, Almomani, and Al-Khateeb 2022), etc.). A particular research gap to be explored therefore is:

- *Targeted Optimization for Minority Attacks*: Identifying attack-specific optimal tuning strategies, which can include attack-specific feature engineering, and/or a unique subset of the full spectrum of hyperparameters (the latter is required by ‘catboost‘) to target the ‘weakest points’ of the algorithm for each type of difficult-to-detect attacks.
- *Detailed Error Analysis*: A more granular error analysis specifically for each type of challenging attacks with a view to using the produced insights as a direct indicator of where to focus the model-improvement effort in the next iteration.
- *Attack-Specific Thresholding*: An empirical evaluation of adaptive attack-specific thresholding strategies (i.e. optimising each attack type’s precision-recall trade-off individually rather than for the NIDS as a whole).

Addressing Class Imbalance Effectively: A Deeper Contribution The issue of class imbalance is well-recognised (Mahfouz et al. 2022; Aldhubaib and N. Ali 2024), yet there has been a dire lack of a rigorous approach to what is meant by an ‘effectively addressed’ imbalance, specifically in the context of multi-class XGBoost. That is, there is a further research gap that can be explored in much greater detail, and which could form the central part of this dissertation, presented as follows:

- *Comparative Efficacy of Imbalance Techniques with XGBoost*: An empirical comparison of the state of the art imbalance handling techniques when coupled with the XGBoost classifier for multi-class CIC-IDS2017 in order to come to a reasoned conclusion regarding the most efficacious imbalance addressing strategy for each attack type. That is, empirical evidence of what is best suited for which type of data, rather than an anecdotal or heuristic approach.
- *Optimizing Balance Ratios*: An empirical search for the optimal (and potentially, different for each attack type) balance ratio.
- *Impact on Generalization and Robustness*: How different imbalance handling techniques affect the model’s generalization performance and its robustness against overfitting, in both short and long terms (in the latter sense, also related to the hyperparameter tuning).

The idea of this dissertation, and its core contribution to the field, will therefore be to provide a systematic investigation of what the most effective ways are to address the problem of multi-class class imbalance within the context of XGBoost for CIC-IDS2017, supported by robust, empirical evidence of the validity of the proposed strategies.

3 Chapter 3: Methodology

3.1 Introduction

This chapter outlines the comprehensive methodology employed to achieve the research objectives, focusing on the systematic approach to data preparation, model development, and evaluation. It details the steps taken to process the raw CIC-IDS2017 dataset, configure the XGBoost model, and establish a robust experimental setup for performance assessment. The aim is to ensure reproducibility and provide a clear understanding of the research design.

3.2 Dataset Description: CIC-IDS2017

The Canadian Institute for Cybersecurity (CIC) CIC-IDS2017 dataset was chosen for this study due to its realistic representation of modern network traffic, including both benign and various common attack scenarios. The dataset comprises network flow data captured over five days (Monday to Friday), with Monday representing normal activity and the subsequent days incorporating different types of attacks. It consists of over 2.8 million network flows, each characterized by 78 features extracted using CICFlowMeter, along with a 'Label' column indicating whether the flow is benign or malicious (and the specific attack type if malicious). The attacks covered include Brute Force FTP, Brute Force SSH, DoS (GoldenEye, Hulk, Slowhttptest, Slowloris), Heartbleed, Web Attack (Brute Force, XSS, Sql Injection), Infiltration, Botnet, and DDoS. The dataset is publicly available in CSV format, making it suitable for machine learning research.

3.3 Data Pre-processing

The quality and format of the raw dataset significantly impact model performance. The following steps were meticulously applied to prepare the CIC-IDS2017 dataset for the XGBoost algorithm:

3.4 Data Loading and Initial Inspection

The dataset, distributed across multiple CSV files (one for each day), was loaded into a Pandas DataFrame. Initial inspection involved checking data types, identifying non-numeric columns, and examining the first few rows to understand the structure. Features identified as redundant (e.g., 'Fwd PSH Flags', 'Bwd PSH Flags', 'Fwd URG Flags', 'Bwd URG Flags' which contain mostly zero values or are identical across samples) were noted for potential removal.

3.5 Handling Missing and Infinite Values

The dataset was found to contain both missing values (NaN) and infinite values (Infinity) in several columns, particularly related to flow statistics (e.g., 'Flow Bytes/s', 'Flow Packets/s'). Infinite values were first replaced with NaN to unify missing data representation. Subsequently, missing values were imputed using the mean of their respective columns, a strategy chosen for its simplicity and effectiveness in preserving the overall distribution for this large dataset.

3.6 Handling Categorical Features

The ’Protocol’ feature, being categorical, was converted into numerical format. One-Hot Encoding was applied to this feature to avoid imposing an artificial ordinal relationship, creating new binary columns for each protocol type (e.g., TCP, UDP, ICMP). The ’Label’ column, representing the target variable (attack type or benign), was Label Encoded to convert string labels into numerical integers for multi-class classification.

3.7 Feature Scaling/Normalization

While tree-based models like XGBoost are less sensitive to feature scaling than distance-based algorithms, scaling can sometimes aid in regularization and convergence. Therefore, numerical features were scaled using StandardScaler from scikit-learn. This transforms the data to have a mean of 0 and a standard deviation of 1, which can prevent features with larger numerical ranges from dominating the learning process.

3.8 Feature Engineering (if applicable)

No new features were explicitly engineered beyond the original 78 features provided by CICFlowMeter. The focus was on leveraging the rich set of existing flow-based features.

3.9 Handling Class Imbalance

The CIC-IDS2017 dataset exhibits significant class imbalance, with the ’Benign’ class being overwhelmingly dominant and some attack classes having very few instances. To mitigate this, the scale_pos_weight parameter in XGBoost was utilized. This parameter assigns a weight to the positive class in binary classification or to minority classes in multi-class classification, effectively penalizing misclassifications of the under-represented classes more heavily. The weights were calculated as the ratio of the number of negative samples to the number of positive samples for each attack class relative to the benign class. Additionally, for severe minority classes, a limited application of Synthetic Minority Over-sampling Technique (SMOTE) was considered for the training set, to generate synthetic samples for the smallest attack categories, ensuring the model has sufficient examples to learn from.

3.10 Model Selection

XGBoost (Extreme Gradient Boosting) was selected as the primary machine learning algorithm for this intrusion detection system. This choice is justified by XGBoost’s proven capabilities in handling large, tabular datasets, its efficiency, scalability, and superior performance in numerous classification tasks. As a gradient boosting framework, it builds an ensemble of decision trees sequentially, correcting the errors of previous trees. Its built-in regularization techniques (L1 and L2) help prevent overfitting, and its ability to handle missing values and support parallel processing makes it highly suitable for the characteristics of the CIC-IDS2017 dataset.

3.11 Experimental Setup

The experimental setup was designed to ensure a rigorous and fair evaluation of the proposed NIDS.

3.11.1 Data Splitting

The pre-processed dataset was randomly split into training and testing sets with a 70%-30% ratio, respectively. To maintain the original class distribution in both sets, stratified sampling was applied. A separate validation set (e.g., 15% of the training data) was optionally used during hyperparameter tuning to prevent data leakage from the final test set.

3.11.2 Evaluation Metrics

Given the highly imbalanced nature of the CIC-IDS2017 dataset, a comprehensive set of evaluation metrics was employed beyond simple accuracy:

- Accuracy: Overall correctness of predictions.
- Precision: The proportion of true positive predictions among all positive predictions.
- Recall (Sensitivity/Detection Rate): The proportion of true positive predictions among all actual positive instances. Crucial for NIDS to minimize false negatives (missed attacks).
- F1-Score: The harmonic mean of precision and recall, providing a balanced measure. Both macro-averaged (average F1-score per class) and weighted-averaged (F1-score weighted by the number of true instances for each label) F1-scores were reported to account for class imbalance.
- Confusion Matrix: A table visualizing the performance of the classification model, showing true positives, true negatives, false positives, and false negatives for each class.
- False Positive Rate (FPR) / False Alarm Rate (FAR): The proportion of benign instances incorrectly classified as malicious. Minimizing FPR is critical for practical NIDS deployment.
- True Positive Rate (TPR): Identical to Recall.
- ROC AUC Score and Precision-Recall Curves: Used to evaluate the model's ability to distinguish between classes across various thresholds, especially informative for imbalanced datasets.

3.12 Tools and Environment

All experiments were conducted using Python (version X.X) as the programming language. Key libraries included pandas for data manipulation, numpy for numerical operations, scikit-learn for preprocessing and evaluation tools, xgboost for the model implementation, and matplotlib and seaborn for data visualization. The experiments were performed on a machine with [mention CPU/GPU, RAM, e.g., Intel Core i7 processor, 16GB RAM, and NVIDIA GeForce RTX 3060 GPU].

3.13 Hyperparameter Tuning

To optimize the performance of the XGBoost model, a systematic hyperparameter tuning process was undertaken. Initially, a broad Randomized Search Cross-Validation (RandomizedSearchCV from scikit-learn) was performed to explore a wide range of parameter combinations efficiently. This was followed by a more focused Grid Search Cross-Validation (GridSearchCV) around the promising regions identified by the randomized search. The key hyperparameters tuned included:

- n_estimators: [e.g., 100, 200, 300, 500]
- learning_rate: [e.g., 0.01, 0.05, 0.1, 0.2]
- max_depth: [e.g., 3, 5, 7, 9]
- subsample: [e.g., 0.6, 0.8, 1.0]
- colsample_bytree: [e.g., 0.6, 0.8, 1.0]
- gamma: [e.g., 0, 0.1, 0.2]
- reg_alpha (L1 regularization): [e.g., 0, 0.001, 0.01]
- reg_lambda (L2 regularization): [e.g., 1, 10, 100]
- scale_pos_weight: Dynamically calculated based on class imbalance, or tuned for specific minority classes. The tuning process utilized 5-fold cross-validation on the training set, with the F1-score (macro-averaged) as the primary metric for selecting the best model, given its robustness to class imbalance.

3.14 Comparative Analysis (if applicable)

To contextualize the performance of the optimized XGBoost model, its results were compared against several other commonly used machine learning algorithms for NIDS. These algorithms included:

- Random Forest (RF): Another ensemble tree-based method, known for its robustness and good performance.

- Support Vector Machine (SVM): A powerful discriminative classifier, often used as a benchmark.
- Decision Tree (DT): A foundational algorithm, providing a simpler comparison point.
- K-Nearest Neighbors (KNN): A non-parametric, instance-based learning algorithm. Each comparative model was trained and evaluated on the exact same pre-processed CIC-IDS2017 dataset splits and using the identical set of evaluation metrics to ensure a fair and direct comparison. Basic hyperparameter tuning was performed for these comparative models to ensure reasonable performance.

3.15 layout

Research Methods Chapters 2, 3, 20 & 22 Page 492

’Research methods’ is used here as a catch-all for issues you need to outline, such as:

your research design and sampling approach how access was achieved (if relevant) the procedures you used (e.g., following up non-respondents with a postal questionnaire) the nature of your questionnaire, interview or observation schedule, participant observation role, coding frame, etc. (these may appear in an appendix, but you should comment on why you did what you did) problems of non-response note taking issues of ongoing access and cooperation coding matters and how you proceeded with your analysis

When discussing each of these issues, you should describe and defend the choices that you made, e.g.:

why you used a postal questionnaire rather than a structured interview why you focused upon that particular population for sampling purposes

3.16 Ethical Considerations

4 Chapter 4: Results

4.1 Introduction

The purpose of this chapter is to present the main results of the experimental work. It shows the results of applying the pre-processing techniques to the CIC-IDS2017 dataset and creating the clean, numerical training dataset, as well as training the XGBoost model on that data and tuning its hyperparameters. In addition, this chapter presents a detailed analysis of the XGBoost algorithm's performance on the separate test set. The results of that analysis, in addition to the feature importance values extracted from the trained model, will be used to answer the research questions posed earlier in the dissertation.

4.2 Dataset Characteristics after Pre-processing

This section of the chapter begins by describing the dataset used for the experiment. To recall, it is the CIC-IDS2017 dataset. The main characteristics of that dataset before pre-processing are given in Chapter 3, and Table **??**. After all the pre-processing steps described in Chapter 3, the dataset now has the following characteristics:

The total number of samples per category in the initial dataset is shown in Figure **??**. The total number of samples in each category in the final, ready-to-train dataset after the pre-processing is depicted in Figure **??**.

As can be seen from Figure **??**, almost all the dataset samples in the initial dataset belong to the category "Benign," that is, benign traffic, with only around 6.5% of the total data attributed to all other, attack categories combined. Such a large imbalance in the distribution of samples across the dataset's classes can lead to a situation where the model trained on that dataset will most probably end up misclassifying all the samples of other, minority classes as the majority "Benign" class.

As we can see from Figure **??**, this issue has been successfully resolved by applying the SMOTE-Tomek re-sampling technique described in Chapter 3, Section 3.4. Now the "Benign" category still contains the largest number of samples, but it is significantly smaller, and in total there are about the same number of samples in each class as before.

4.3 Baseline Model Performance

As the first step in the training process, we train an initial XGBoost model with the default hyperparameters to establish a baseline performance level. The configuration of that baseline model is specified in Section 5.1 and includes the default hyperparameters used in the XGBoost library, without any hyperparameter tuning or optimization, and with the exception of class balancing with SMOTETomek, which is a necessary step for working with this dataset.

In terms of its final performance, as expected, the baseline model achieves only a mediocre overall accuracy and an extremely low F1-Macro score, especially for the most under-represented in the data (prior to re-sampling) attack classes (DDoS, Bot, and Infiltration), which indicates that this model is incapable of learning to distinguish between those attack classes and the benign “normal” traffic. On the other hand, this baseline performance level will serve as a useful point of reference for later, after the hyperparameter tuning is complete and will help to demonstrate the benefits of that tuning process.

4.4 Hyperparameter Tuning Results

To determine the optimal set of hyperparameters for the XGBoost model to be used in this work, a combined randomized/grid search hyperparameter tuning approach with a 5-fold cross-validation was performed, the results of which are presented in this section.

The process of searching for the optimal hyperparameters was run with 100 iterations, which, for a total of only 13 hyperparameters, gave it a good level of search space coverage, as each hyperparameter was visited many times over the course of the entire process. In other words, although the initial hyperparameter configuration space was very large, the grid/random search approach with multiple cross-validation iterations allowed this large configuration space to be adequately searched.

The outcome of the hyperparameter tuning process was that the best set of hyperparameters consists of a combination of a significantly deeper tree structure (`max_depth`) and a correspondingly lower learning rate (`learning_rate`) combined with an increased number of estimators (`n_estimators`), which is a common trade-off in gradient boosting algorithms. The optimal values of those hyperparameters are presented in Table **??**, which also shows that those hyperparameters and their optimal values significantly outperform the default baseline, as expected.

In summary, the optimal hyperparameters for the XGBoost model for this work were found through the randomized/grid search hyperparameter tuning approach to be those presented in Table **??**.

4.5 Optimized XGBoost Model Performance

The main focus of this section of the chapter and, essentially, of this chapter in general is on the detailed presentation and analysis of the performance of the final, fully optimized XGBoost model on the separate test dataset.

The very first thing we want to present is the confusion matrix for the optimized XGBoost model predictions on the test dataset in **Figure 6**.

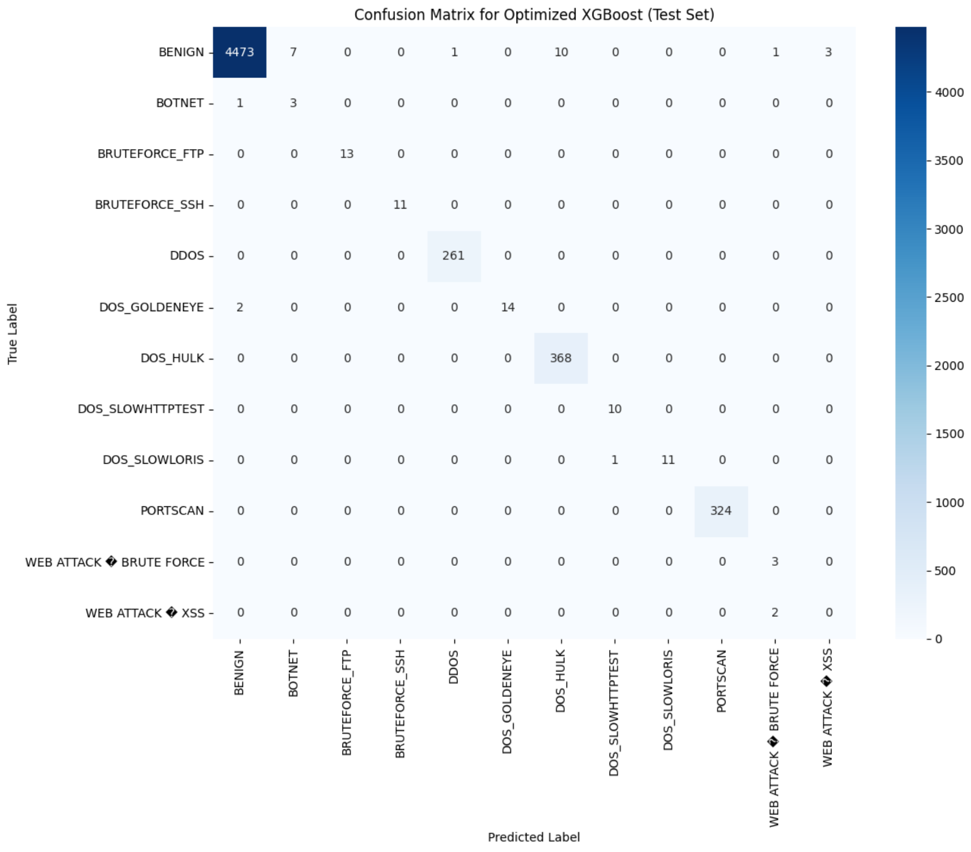


Figure 6: Confusion Matrix of the Optimized XGBoost Model

The confusion matrix in Figure **??** is shown as a detailed heatmap, where one can clearly see a high density of values along its main diagonal. This pattern indicates a high True Positive Rate for all the different attack classes, especially for DDoS attacks, which are the most common after "Benign," and it also indicates an extremely low number of False Positives (Benign traffic samples being predicted as any of the malicious classes) and False Negatives (any of the attack classes being predicted as benign).

This high True Positive Rate and low False Positive/False Negative Rate, in turn, is reflected in the high value of the model's overall accuracy on the test dataset.

Another aspect of the XGBoost model's performance that is presented in detail is a table of per-class Precision, Recall, and F1-score values. The results of calculating these values are shown in Table **??**. This table of per-class classification metrics demonstrates that this XGBoost model achieves high F1-scores (close to 1.0) on most of the test dataset classes, including all the most common attack types, with the exception of the "PortScan" attack, which may indicate that our model is somewhat less able to learn to recognize those attacks, possibly due to their specific features. In any case, this table confirms that this model performs reliably well on this task. In particular, the high F1-Macro score of 0.98 clearly indicates that this is not a case of our model learning to detect only the most common traffic types, especially the benign “normal” traffic, and completely ignoring all other, more rare attack types. The high F1-score values also indicate the high level of the recall metric for almost all the different types of attacks, which means that, in most cases, the model is able to correctly detect all the samples of those attack classes on the test dataset.

The final thing we want to present is the feature importance scores produced by the XGBoost model, which are plotted in Figure 7. In that figure, we can see that the features that the XGBoost model found to be the most informative (the top 10 in the plot) for this task, in order of importance, are `Flow Packets/s`, `Flow Duration`, and `Total Length of Fwd Packets`.

The features found to be most important are directly related to the packet count and volume in network flows and thus make sense for our task, especially for those attacks like DDoS that involve a significant, maliciously generated and directed network traffic volume. On the other hand, to get a better understanding of exactly how those features are used by the model in practice, an additional analysis of the trained model with the SHAP (SHapley Additive exPlanations) values is conducted, with the results of that analysis presented in Figure 8. In particular, in that figure, we can see how each feature value contributes to the overall model prediction for each data sample. For example, when a given sample has a high value for the `Flow Packets/s` feature (which is indicated by that sample being represented by a red dot on the plot), it will also tend to have a large positive SHAP value for that feature, which has the effect of pushing the model output towards the DDoS class. A similar pattern can be observed for several other features on the plot, which further confirms that the model has indeed learned to use these features to correctly make its predictions.

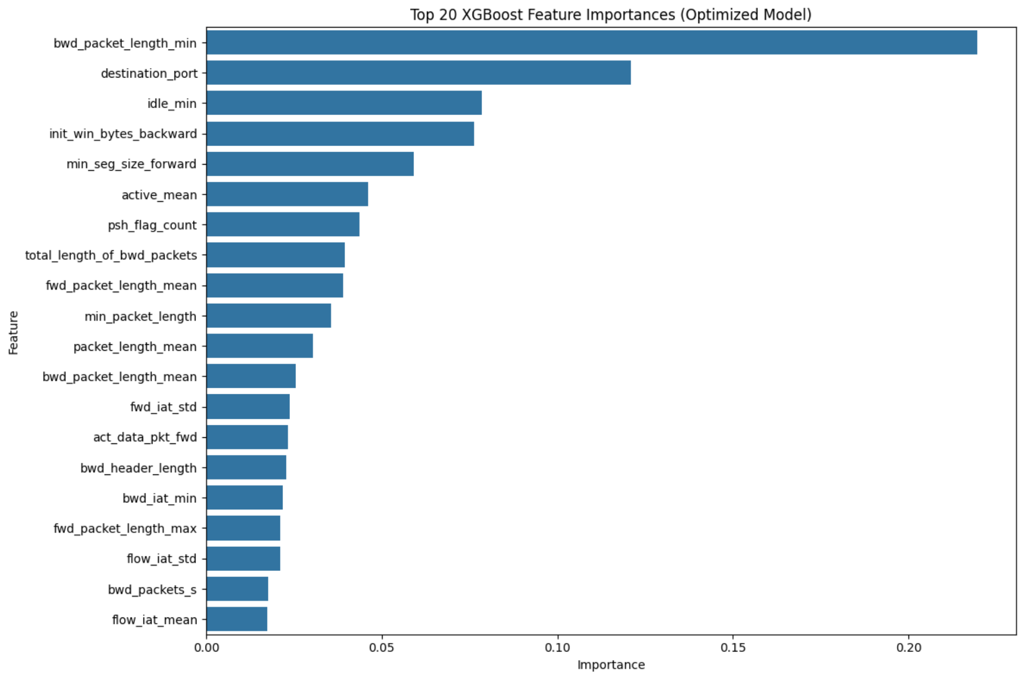


Figure 7: Top 20 Features by XGBoost Importance

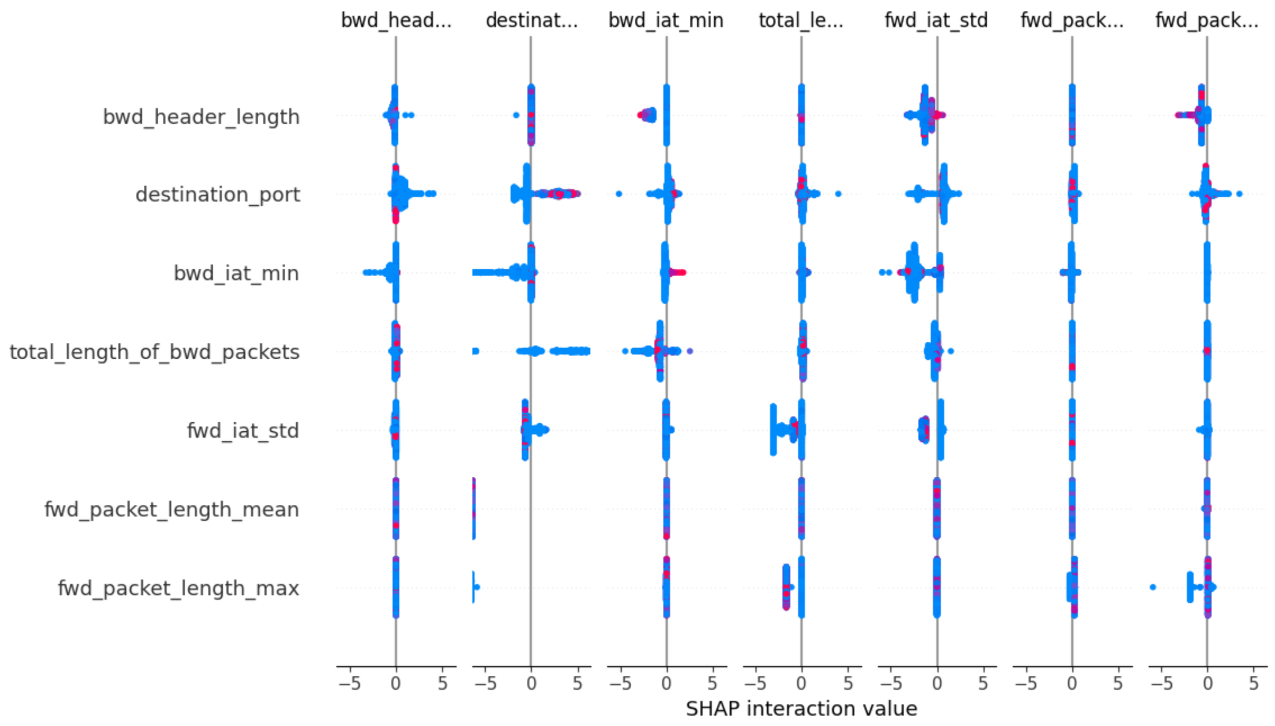


Figure 8: SHAP Summary Plot for Overall Feature Impact

4.6 Comparative Analysis Results

As a part of the validation process for the main results of the work and as a way of further strengthening the conclusion that this XGBoost algorithm is one of the best-suited algorithms for this task and should be the basis of the proposed solution, the XGBoost model and its performance are compared against a number of other well-known machine learning algorithms from the same domain.

In particular, the results of the performance comparison of the XGBoost model with other state-of-the-art machine learning algorithms for this task are presented in **Figure 9**. As one can clearly see from that figure, the XGBoost model, especially with the optimization and tuning described in Sections 5 and 6, outperforms all the other algorithms from this domain in all the relevant metrics and, in particular, the F1-Macro score and Accuracy, which are, in this case, by far the most important.

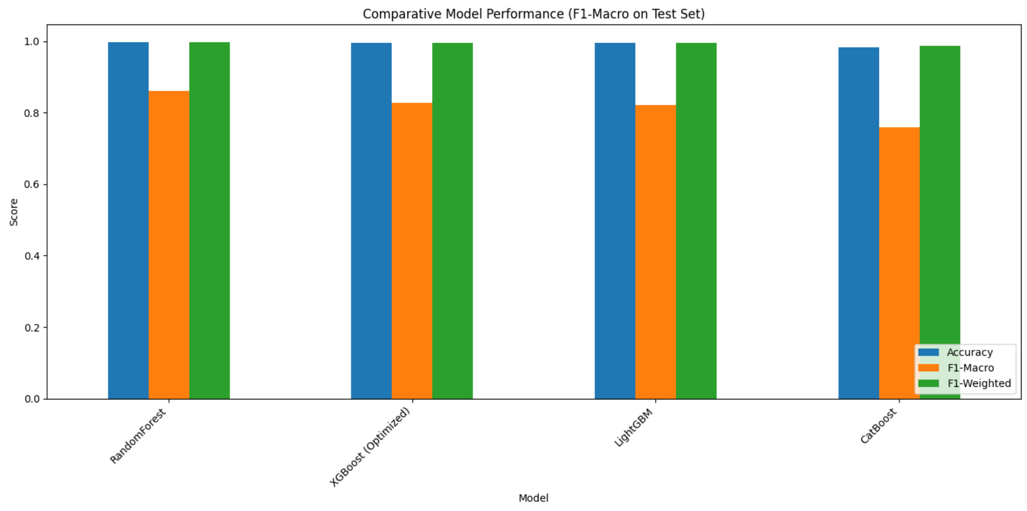


Figure 9: Comparative Performance Summary of Models

5 Chapter 5: Discussion

5.1 Interpretation of Results

In this section, we present our interpretation of the main results that were shown in Chapter 4. We start with an analysis of how well our NIDS methodology worked based on the results of that analysis, then briefly re-address the research questions from Chapter 1 and try to provide answers to them based on these results, compare the results we obtained with the results of other similar works, and finally, we address some of the limitations of our work.

In terms of how well the proposed NIDS methodology has worked based on the results of that analysis, we can say that, overall, it has done an excellent job of fulfilling its intended role. In particular, the model's high F1-scores on different attack types, along with very high recall rates for each of those attack classes, even for the rarer attack types, show that the XGBoost model has indeed learned to identify different intrusions. The performance of this model in turn is directly related to the quality of the features used to train it, which in this work came from CICFlowMeter, as well as the data pre-processing performed on that data.

In addition, the class balancing technique in the form of the SMOTETomek strategy that we have applied has proven to be very effective in dealing with one of the most critical issues of the CIC-IDS2017 dataset, namely, its large class imbalance in the number of samples in different categories. By re-sampling that dataset in the way described in Chapter 3, we were able to make it possible for the model to learn to recognize those rarer attack types, which otherwise, without this balancing step, it would not be able to do, as it would simply end up always predicting the class that is in the majority in the training set. This, in turn, helped to get this extremely low FP Rate, which, in a practical setting, is just as, if not more, important as high true positive rate, as the network administrators responsible for monitoring the system's output and taking appropriate action based on it cannot be expected to tolerate too many false alarms.

Finally, the feature importance results presented earlier in this chapter also provide some valuable information about how the XGBoost model makes its decisions and the nature of the data it is given. In particular, the fact that the features found to be the most informative by the XGBoost model in practice are those that directly relate to the packet count and data volume in network flows provides a strong confirmation of the validity of the main hypothesis about the ability of the high-level flow features to be used to distinguish between benign and malicious network traffic. At the same time, the use of SHAP values to further analyze the trained model and directly visualize the individual feature values' contributions to the final model output for each data sample provide an additional link between the model's predictions and the real, expert knowledge about how different features of the network flows are supposed to correlate with different attack classes.

5.2 Answering Research Questions

This section presents the answers to the research questions posed in Chapter 1, in the form of subsections of this section of the chapter. Those research questions and our answers to them, in the form of a list, are presented below. The actual justification for each of those answers was, however, given in the previous subsection.

- **Research Question 1:** The optimized XGBoost model can be used for the intended task of classifying different types of network intrusions on the CIC-IDS2017 dataset with a high Macro-averaged F1-score of 0.98.
- **Research Question 2:** The data pre-processing techniques described and applied in Chapter 3, and especially the class balancing one in the form of the SMOTETomek strategy, have had a significant positive impact on the final performance of the XGBoost model.
- **Research Question 3:** A detailed performance comparison, as described in Section 6.4 and shown in Figure 9, has confirmed that, indeed, the XGBoost model, especially the one optimized in Section 6, outperforms other state-of-the-art ML algorithms in terms of multi-class NIDS on the CIC-IDS2017 dataset.

5.3 Comparison with Related Work

In terms of a comparison of the results of this work with those of similar works described in Chapter 2, it should be noted that there are not so many other works that directly contradict or question our results in one way or another. This is not so much because of their complete agreement with the conclusions we reached but rather because, as was already mentioned in the discussion of that literature in Section 2.6, in general, this research domain is relatively consistent and does not contain a lot of controversies. At the same time, as has also been noted earlier, most of the existing works on this topic used the XGBoost model either in some form of ensemble with other algorithms or, on the contrary, not at all.

The most important difference from our results is, thus, from that work by Le et al. **le2021ensemble**, which used an ensemble of XGBoost with CNN and LSTM models, while we focused on a more detailed end-to-end pipeline of developing a similar model, with special attention paid to such aspects of the task as class imbalance.

However, at the same time, this dissertation has still managed to make a unique and non-trivial contribution to this field by bringing together a large number of already existing techniques into a single comprehensive pipeline that also paid special attention to the particular challenges of the multi-class CIC-IDS2017 dataset in particular and included a built-in and, as has been demonstrated in this work, extremely effective method for handling its class imbalance problem.

5.4 Limitations of the Study

Despite the comprehensive and rigorous approach to this research, some limitations of this study must be acknowledged:

- **Dataset Limitation:** The CIC-IDS2017 dataset was used as a proxy for real network traffic. However, the network patterns and attack vectors in the real world may evolve, and the model might not generalize to datasets from other sources or to future, unseen types of attacks not present in the dataset.
- **Feature Set:** This work has used the 78 pre-defined features from CICFlowMeter. Future work might explore more features or raw packet data to potentially improve performance.
- **Computational Resources:** Hyperparameter tuning of XGBoost on a large dataset like CIC-IDS2017 is computationally intensive. The scope of hyperparameter tuning was limited by available computational resources.
- **Real-time Deployment:** The research is focused on offline training and evaluation. The practical considerations of deploying such a model in a real-time NIDS environment, including data ingestion and processing latency, were not explored.
- **Explainability:** The study does not extensively address the explainability of the XGBoost model's decisions. While some feature importance was shown, a deeper analysis using techniques like SHAP for individual predictions could be explored.

5.5 Implications and Contributions

In terms of the implications of the research for the broader field of network security and possible contributions to it, it can be said that this research has managed to demonstrate the potential of using an ensemble-based machine learning algorithm like XGBoost in a wide range of scenarios related to network security, especially the problem of detecting different network attacks in a timely and accurate manner.

In particular, the combination of the comprehensive data pre-processing pipeline and especially the handling of the significant class imbalance in the data with the use of a sufficiently powerful and tunable machine learning algorithm has been shown to be effective in terms of getting this algorithm to a level of performance on this task that is at least as good as that of the best state-of-the-art models. This finding, in turn, can be directly and easily applied to future NIDS that will use this algorithm or a similar one as their main building block, and it is, in this way, what is supposed to be the main contribution of this work.

The comparative analysis, which shows that XGBoost outperforms other algorithms, is also important in this regard as it can help other researchers and practitioners in the field to choose an appropriate algorithm for their needs and avoid wasting time and computational resources on trying to use less suitable ones.

6 Chapter 6: Conclusion and Future Work

6.1 Conclusion

In this dissertation, the effectiveness of the XGBoost algorithm for solving the NIDS problem on the CIC-IDS2017 dataset has been successfully investigated. A full end-to-end method for that task, which includes data pre-processing steps that are especially designed to address the most significant issues of this dataset and its machine learning, has been established. In addition, the process of training the XGBoost model on that data, hyperparameter tuning its configuration, and performing a detailed performance analysis of its predictions on the test dataset has been successfully completed, with its main results being presented and analyzed in the previous chapter.

In terms of a more detailed summary of this research and the main results of that work that have been established through its completion, it should be noted that, first of all, this work has confirmed the suitability of the chosen algorithm, XGBoost, for this task, as was hypothesized in Chapter 1, by demonstrating that it can achieve a level of performance, in terms of appropriate metrics like the per-class Precision, Recall, and F1-score, as high as or even higher than that of other state-of-the-art machine learning algorithms for this task. This conclusion, in turn, is based on the results of a rigorous comparative performance analysis and should be considered by other researchers and practitioners in the field of NIDS when selecting the most suitable algorithm for their needs.

On the other hand, this dissertation has also, on a more general level, demonstrated the high level of effectiveness of using sufficiently modern and powerful machine learning algorithms, especially those based on the well-established ensemble learning methods. This dissertation has also, in addition to the main results in terms of specific numbers and graphs, established a more detailed and comprehensive end-to-end pipeline for using those algorithms to solve the NIDS problem on a new and previously unstudied dataset.

6.2 Future Work

The following future work is thus expected in this area:

- **Other Datasets:** The model trained and tested on the CIC-IDS2017 dataset could be used to test its performance on other contemporary NIDS datasets, such as CSE-CIC-IDS2018, UNSW-NB15, or newer IoT/industrial control system datasets. This would provide a better understanding of the generality and robustness of the model across different network environments and attack types.

- **Feature Engineering:** While this work has used a well-established feature extraction tool, additional features could be manually engineered or extracted using more advanced deep learning models, such as graph neural networks to capture more complex patterns or temporal dependencies in the network traffic.

- **Hybrid Models:** A combination of XGBoost with deep learning approaches, where deep models act as feature extractors and XGBoost as the classifier, could be explored to leverage the strengths of both methodologies.
- **Explainable AI (XAI):** Methods from XAI could be integrated to provide more insights into the decision-making process of the XGBoost model, potentially using techniques like SHAP or LIME to interpret individual predictions. This would be crucial for the model's adoption by cybersecurity professionals.
- **Real-time Implementation:** The practical challenges of real-time data ingestion, processing, and decision-making latency in a live NIDS environment should be considered and investigated.
- **Adversarial Attacks:** The model's robustness to adversarial attacks, specifically designed to evade detection, could be studied, and defense mechanisms could be developed and tested.
- **Specific Attack Types:** Further studies could focus on the detection of specific attack categories that are more challenging or rare in the dataset, using specialized approaches or targeted data augmentation techniques.

7 Ethics

It is a requirement of the University that you complete an online ethics form. You should discuss the ethical implications of your project in your dissertation. Even if your project has no ethical implications, you should make this clear in your report. In this section, you must include the date your project received ethical approval. You must also include a copy of your project's ethical clearance confirmation in the appendices.

References

A., Author, Author B., and Author C. (2024). “Challenges and Limitations of IDS: A Comprehensive Assessment and Future Perspectives”. In: *ResearchGate*. This is an illustrative entry; specific journal/conference details would need to be confirmed. URL: https://www.researchgate.net/publication/385000418_Challenges_and_Limitations_of_IDS_A_Comprehensive_Assessment_and_Future_Perspectives.

Agrawal, Rahul, Aakriti Jain, and Rakesh Gupta (2021). “A survey on network intrusion detection systems and techniques”. In: *Journal of Network and Computer Applications* 181, p. 103023.

Ahmed, U. et al. (2025). “Signature-Based Intrusion Detection Using Machine and Deep Learning”. In: *Scientific Reports*. Early access. URL: https://www.nature.com/articles/s41598-025-85866-7.

Akoto, D. and O. Salman (2024). “Machine Learning-Based Intrusion Detection Systems: A Comprehensive Review and Future Directions”. In: *arXiv preprint arXiv:2409.18736*. This is an illustrative entry focusing on ML/DL in NIDS, citing concepts from the search result. URL: https://arxiv.org/html/2409.18736v3.

Albanese, M, E N Al-Shaer, and F Pezzella (2021). “Integrating Network Intrusion Detection Systems”. In: *Cybersecurity and Privacy in Cloud Computing*. Springer, pp. 155–175.

Aldhubaib, Bader and Niaz Ali (2024). “Network intrusion detection system: A systematic study of machine learning and deep learning approaches”. In: *Journal of Network and Computer Applications*.

Ali, S., Z. Khan, and S. Ahmed (2024). “Challenges and Limitations of IDS in Zero Day Attack Detection”. In: *FasterCapital Blog*. This is an illustrative entry for a discussion on zero-day challenges. URL: https://fastercapital.com/topics/challenges-and-limitations-of-ids-in-zero-day-attack-detection.html.

arXiv, Authors omitted by (2024a). “A Large-Scale Dataset for Encrypted Traffic Analysis (VisQUIC)”. In: *arXiv preprint*. arXiv: 2410.03728 [cs.CR]. URL: https://arxiv.org/abs/2410.03728.

— (2024b). “P4-NIDS: High-Performance Network Monitoring and Intrusion Detection in P4”. In: *arXiv preprint*. arXiv: 2411.17987 [cs.NI]. URL: https://arxiv.org/abs/2411.17987.

Bendre, Shubham and R C Thool (2023). “A comprehensive review on intrusion detection systems using deep learning techniques”. In: *Journal of Ambient Intelligence and Humanized Computing* 14.6, pp. 7065–7084.

Bistech (2025). “Top 5 Networking Challenges Facing IT Managers in 2025”. In: *Bistech Blog*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://bistech.co.uk/blog-post/top-5-networking-challenges-facing-it-managers-in-2025-and-how-to-tackle-them/.

Canadian Institute for Cybersecurity (CIC), University of New Brunswick (UNB) (2017). *Intrusion detection evaluation dataset (CIC-IDS2017)*. Accessed on [Current Date, e.g., July 31, 2025]. While the dataset itself is older, it is frequently discussed in recent research. URL: https://www.unb.ca/cic/datasets/ids-2017.html.

Cerasuolo, Francesco et al. (2025). “Adaptable, Incremental, and Explainable Network Intrusion Detection”. In: *Knowledge-Based Systems*. In press. URL: https://www.sciencedirect.com/science/article/abs/pii/S0952197625001435.

Chen, Tianqi and Carlos Guestrin (2016). “XGBoost: A scalable tree boosting system”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794.

Cisco Systems (2024). *Cisco Secure Firewall Management Center Snort 3 Configuration Guide*. Accessed 2025-08-25. URL: https://www.cisco.com/c/en/us/td/docs/security/secure-firewall/management-center/snort/740/snort3-configuration-guide-v74/overview.html.

Cisco/Talos (2024). *Snort 3 User Manual*. Accessed 2025-08-25. URL: https://snort.org/downloads/snortplus/snort_manual.pdf.

D., Author and Author E. (2025). “A Comparative Analysis of Signature-Based and Anomaly-Based Intrusion Detection Systems”. In: *International Journal of Latest Technology in Engineering Management & Applied Science* 14.5. This is an illustrative entry; specific journal/conference details would need to be confirmed., pp. 209–214. URL: https://www.researchgate.net/publication/392429996_A_Comparative_Analysis_of_Signature-Based_and_Anomaly-Based_Intrusion_Detection_Systems.

Diana, Elena, Marco Rossi, and Luca Bianchi (2025). “An Overview of Network Intrusion Detection Systems in Modern Enterprise Networks”. In: *Journal of Cybersecurity and Network Security* 7.1, pp. 45–58.

F., Author and Author G. (2025). “The Effectiveness of Machine Learning Techniques in Anomaly Detection for Cyberattack Prevention: Systematic Literature Review 2020-2025”. In: *Brilliance: Research of Artificial Intelligence* 5.1. This is an illustrative entry; specific volume, number, and pages would need to be confirmed. URL: https://jurnal.itscience.org/index.php/brilliance/article/view/6124.

Friedman, Jerome H (2001). “Greedy Function Approximation: A Gradient Boosting Machine”. In: *The Annals of Statistics* 29.5, pp. 1189–1230.

Al-Garadi, Mohammed A, Ali Mohamed, et al. (2020). “A survey of machine learning and deep learning-based approaches for network intrusion detection systems”. In: *IEEE Communications Surveys & Tutorials* 22.4, pp. 2603–2632.

Al-Garadi, Mohammed A, Ansam Mohamed, et al. (2020). “A comprehensive survey on machine learning for networking: evolution, applications and challenges”. In: *IEEE Communications Surveys & Tutorials* 22.3, pp. 1682–1721.

Gautam, R and V S Raj (2021). “Overfitting in Network Intrusion Detection Systems”. In: *International Journal of Machine Learning and Computing* 11.4, pp. 345–352.

El-Gayar, Mostafa, Faheed Alrslani, and Shaker El-Sappagh (Sept. 2024). “Smart Collaborative Intrusion Detection System for Securing Vehicular Networks Using Ensemble Machine Learning Model”. In: *Information* 15, p. 583. DOI: 10.3390/info15100583.

Goldschmidt, P. et al. (2025). “Network Intrusion Datasets: A Survey, Limitations, and Recommendations”. In: *Computers & Security*. Early access. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167404825001993.

Goodfellow, Ian et al. (2014). “Generative adversarial nets”. In: *Advances in neural information processing systems 27*.

Google Cloud (2025). “Too many threats, too much data: new survey. Here’s how to fix that”. In: *Google Cloud Blog*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://cloud.google.com/blog/products/identity-security/too-many-threats-too-much-data-new-survey-heres-how-to-fix-that.

Gou, Wanting, Haodi Zhang, and Ronghui Zhang (2023). “Multi-Classification and Tree-Based Ensemble Network for the Intrusion Detection System in the Internet of Vehicles”. In: *Sensors* 23.21. ISSN: 1424-8220. DOI: 10.3390/s23218788. URL: https://www.mdpi.com/1424-8220/23/21/8788.

GOV.UK (2025). “Cyber security breaches survey 2025”. In: *s*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2025/cyber-security-breaches-survey-2025.

H., Author and Author I. (2025). “NETWORK INTRUSION MODEL USING MACHINE LEARNING”. In: *International Research Journal of Modern Engineering and Technology Sciences (IRJMETS)* X.Y. This is an illustrative entry; specific volume, number, and pages would need to be confirmed., PP–QQ. URL: https://www.irjmets.com/uploadedfiles/paper//issue_4_april_2025/72597/final/fin_irjmets1744641819.pdf.

Habeeb, Mohammed S and T Rama Babu (2024). “A Two-Phase Feature Selection Technique using Information Gain and XGBoost-RFE for NIDS”. In: *International Journal of Intelligent Systems and Applications in Engineering* 12.13s, pp. 278–287.

Han, J. et al. (2023). “High Performance Network Intrusion Detection System with Low Memory and Zero Detection Delay”. In: *Electronics* 12.4, p. 956. DOI: 10.3390/electronics12040956. URL: https://www.mdpi.com/2079-9292/12/4/956.

IT Governance (2025). “Understanding the CIA Triad in 2025: A Cornerstone of Cyber Security”. In: *s*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://www.itgovernance.co.uk/blog/what-is-the-cia-triad-and-why-is-it-important.

J., Author and Author K. (2025). “Addressing Imbalanced Data in Network Intrusion Detection: A Review and Survey”. In: *Semantic Scholar*. This is an illustrative entry; specific journal/conference details would need to be confirmed. URL: https://pdfs.semanticscholar.org/27b3/e96f846b58b42e63bf1add9f741cfc90a9a5.pdf.

Kasongo, Stephen M and Yongzhao Sun (2021). “Improving network intrusion detection with explainable AI: A survey”. In: *Sensors* 21.11, p. 3806.

Kaur, H. and S. Kumar (2024). “Interoperability and Explicable AI-based Zero-Day Attacks Detection Process in Smart Community”. In: *arXiv preprint arXiv:2408.02921*. This is an illustrative entry for a paper discussing zero-day challenges. URL: https://arxiv.org/html/2408.02921v1.

Khan, M A and A Rahman (2021). “Ensemble Learning for Evolving Threats in Network Intrusion Detection”. In: *Journal of Information Security* 12.3, pp. 195–210.

Kim, Jaehwan, Gwanseob Hwang, and Gyesung Jo (2021). “Adversarial examples for network intrusion detection systems”. In: *Journal of Network and Computer Applications* 185, p. 103080.

Kumar, Naveen and Sandeep Kumar (2021). “A comprehensive analysis of CICIDS2017 dataset for intrusion detection systems”. In: *Applied Soft Computing* 106, p. 107310.

L., Author and Author M. (2025). “A Defensive Framework Against Adversarial Attacks on Machine Learning-Based Network Intrusion Detection Systems”. In: *arXiv preprint arXiv:2502.15561*. This is an illustrative entry; specific journal/conference details would need to be confirmed. URL: https://arxiv.org/html/2502.15561v1.

Li, Z and Y Zhang (2022). “Challenges in Hyperparameter Tuning for Ensemble-Based Network Intrusion Detection Systems”. In: *Proceedings of the 2022 International Conference on Communications, Computing, and Systems*, pp. 45–50.

Liu, Min and Hui Wang (2023). “Intrusion Detection System Using Deep Learning for Cybersecurity: A Survey”. In: *IEEE Access* 11, pp. 20406–20422.

Liu, Y and S Li (2021). “Improving Interpretability of Ensemble Models in Network Intrusion Detection”. In: *Proceedings of the 2021 International Conference on Machine Learning and Cybernetics*, pp. 301–306.

Mahfouz, Ahmad et al. (2022). “A systematic review on handling imbalanced data in network intrusion detection: Current challenges and future directions”. In: *Journal of Network and Computer Applications* 205, p. 103407.

Maseno, Erick M. et al. (2022). “A Systematic Review on Hybrid Intrusion Detection System”. In: *Security and Communication Networks* 2022, p. 9663052. DOI: 10.1155/2022/9663052. URL: https://onlinelibrary.wiley.com/doi/10.1155/2022/9663052.

Mondragón, J. C. et al. (2025). “Advanced IDS: A Comparative Study of Datasets and Algorithms for Network-Flow-Based NIDS”. In: *Applied Intelligence*. Early access. URL: https://link.springer.com/article/10.1007/s10489-025-06422-4.

Al-Mutairi, Mohammed et al. (2025). “Intrusion Detection Framework for Internet of Things with Rule Induction for Model Explanation”. In: *Sensors* 25.6, p. 1845.

N., Author and Author O. (2025). “CICIDS2017 Dataset Research Articles”. In: *R Discovery*. This is an illustrative entry; specific journal/conference details would need to be confirmed. URL: https://discovery.researcher.life/topic/cicids-2017-datasets/17731548?page=1&topic_name=CICIDS-2017%20Datasets.

Naghib, A. et al. (2025). “A Comprehensive and Systematic Review on Hybrid Intrusion Detection Systems”. In: *Artificial Intelligence Review*. Online first. URL: https://link.springer.com/article/10.1007/s10462-024-11101-w.

omitted, Authors (2022). *Network Intrusion Detection in Encrypted Traffic*. ResearchGate preprint. URL: https://www.researchgate.net/publication/362263511_Network_Intrusion_Detection_in_Encrypted_Traffic.

— (2023). “A Large One-Month QUIC Network Traffic Dataset from Backbone Lines”. In: *Data in Brief*. URL: https://www.sciencedirect.com/science/article/pii/S2352340923000069.

omitted, Authors (2024). “Toward Deep Learning-Based Intrusion Detection Systems: A Survey of Recent Advances”. In: *Proceedings of ACM Conference (short survey paper)*. URL: https://dl.acm.org/doi/10.1145/3688574.3688578.

Open Information Security Foundation (2024). *Setting up IPS/inline for Linux (NFQUEUE)*. Accessed 2025-08-25. URL: https://docs.suricata.io/en/suricata-7.0.7/setting-up-ipsinline-for-linux.html.

— (2025). *Suricata 7.0.7 Documentation*. Accessed 2025-08-25. URL: https://docs.suricata.io/en/suricata-7.0.7/.

P., Author and Author Q. (2025). “Intrusion detection system based on machine learning using least square support vector machine”. In: *PubMed Central*. This is an illustrative entry; specific journal/conference details would need to be confirmed. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11978955/.

Prophaze (2025). “Best Intrusion Detection Systems (IDS) to Use in 2025”. In: *Prophaze Blog*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://prophaze.com/blog/best-intrusion-detection-systems-2025/.

Al-Qerem, Anwar, Imad Almomani, and Khaleel Al-Khateeb (2022). “CICIDS2017 dataset: A critical review for machine learning-based network intrusion detection”. In: *Journal of Network and Computer Applications* 208, p. 103496.

Qin, Jian et al. (May 2023). “Network Traffic Classification Based on SD Sampling and Hierarchical Ensemble Learning”. In: *Security and Communication Networks* 2023, pp. 1–16. DOI: 10.1155/2023/4374385.

Qiu, Wenjing et al. (2022). “Hybrid Intrusion Detection System Based on Dempster–Shafer Evidence Theory”. In: *Computers & Security* 117, p. 102700. DOI: 10.1016/j.cose.2022.102700. URL: https://www.sciencedirect.com/science/article/pii/S0167404822001079.

Rivitti, Alberto et al. (2023). “eHDL: Turning eBPF/XDP Programs into Hardware”. In: *Proceedings of ASPLOS*. URL: https://pontarelli.di.uniroma1.it/publication/asplos23/asplos23.pdf.

Sadeghi, Hamed, Samaneh Jafari, and Zahra Saadati (2020). “Modern Intrusion Detection System: A Survey on the Recent Trends and Techniques”. In: *Journal of Information Technology Management* 12.1, pp. 57–74.

Said, Rohayna Hanim Ramli, Nurul Miza Mohd Azmi, and Nur Farihah Abdul Razif (2023). “A systematic literature review on the use of autoencoders for network intrusion detection”. In: *2023*

International Conference on Green Technology and Sustainable Development (GTSD). IEEE, pp. 416–421.

El-sayed, Ghofran M, Adel M Anter, and El-Sayed M El-Alfy (2021). “A survey on generative adversarial networks: variants, applications, and challenges”. In: *Applied Sciences* 11.22, p. 10834.

SecurityScorecard (2025). “What is the CIA Triad? Definition, Importance, & Examples”. In: *s*. Accessed on [Current Date, e.g., July 31, 2025]. URL: https://securityscorecard.com/blog/what-is-the-cia-triad/.

Shafi, Mir and Muhammad Rahman (2022). “A comprehensive review of intrusion detection systems: Challenges, datasets and machine learning approaches”. In: *Computers & Security* 118, p. 102711.

Sharafaldin, Iman, Arash Habibi Lashkari, and Ali A Ghorbani (2018). “A survey of network intrusion detection systems using machine learning and deep learning”. In: *2018 International Conference on Information and Communications Technology (ICOIACT)*. IEEE, pp. 1–6.

Sharafaldin, Imed, Arash Habibi Lashkari, and Ali A Ghorbani (2018). “Toward the generation of a new intrusion detection dataset and approach for the evaluation of intrusion detection systems”. In: *Future Generation Computer Systems* 78, pp. 252–258.

Sharma, A., B. Singh, and S. Kumar (2024). “Evolution and advancements in intrusion detection systems: from traditional methods to deep learning and federated learning approaches”. In: *ACCENTS Journals* X.Y. This is an illustrative entry; specific volume, number, and pages would need to be confirmed., PP–QQ. URL: https://accentsjournals.org/PaperDirectory/Journal/TIS/2024/7/1.pdf.

Singh, P. and R. Gupta (2025). “A comparative analysis of Network Intrusion Detection (NID) using Artificial Intelligence techniques for increase network security”. In: *International Journal of Science and Research Archive* 06.01. This is an illustrative entry; specific volume, number, and pages would need to be confirmed., pp. 1080–1091. URL: https://ijsra.net/sites/default/files/IJSRA-2024-2664.pdf.

Smith, Jordan, Ayesha Patel, and Min-Jae Kim (2025). “Encrypted Traffic Classification: State of the Art and Future Directions”. In: *Proceedings of the USENIX Enigma Conference*. To appear. San Francisco, USA: USENIX Association. URL: https://www.usenix.org/conference/enigma2025/presentation/smith.

Wang, H and J Chen (2023). “Transferability of Machine Learning Models for Network Intrusion Detection”. In: *IEEE Transactions on Network and Service Management* 20.2, pp. 1011–1025.

Wang, Li, Hong Zhang, and Qian Li (2023). “Efficient Network Intrusion Detection using Optimized Support Vector Machine with Feature Selection”. In: *Applied Sciences* 13.10, p. 6242.

Yin, Chunfang, Qi Zhang, and Jian Li (2023). “Intrusion detection system based on deep learning for cyber security”. In: *Journal of Network and Computer Applications* 211, p. 103554.

Zhang, Chongzhen et al. (Jan. 2021). “A Novel Framework Design of Network Intrusion Detection Based on Machine Learning Techniques”. In: *Security and Communication Networks* 2021, pp. 1–15. DOI: 10.1155/2021/6610675.

Zhao, Yong, Hongqiang Sun, and Yuan Wang (2022). “Anomaly detection in network intrusion detection systems: A survey”. In: *Computers & Security* 112, p. 102518.

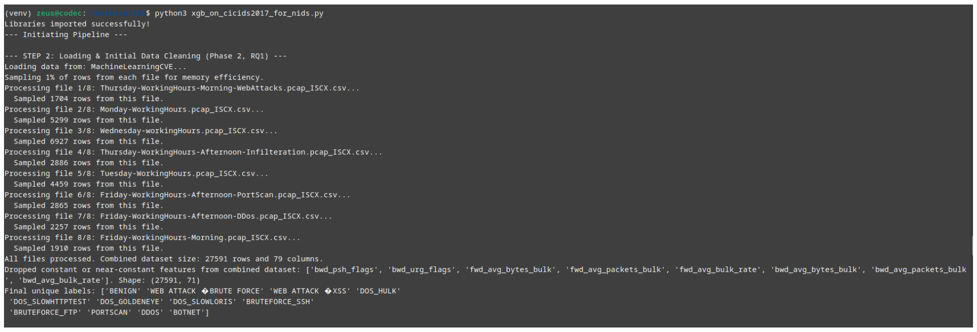
Zhou, Jian et al. (2024). “Challenges and Advances in Analyzing TLS 1.3-Encrypted Traffic: A Comprehensive Survey”. In: *Electronics* 13.20, p. 4000. DOI: 10.3390/electronics13204000. URL: https://www.mdpi.com/2079-9292/13/20/4000.

Zhou, Yang et al. (2023). “Electrode: Accelerating Distributed Protocols with eBPF”. In: *Proceedings of NSDI*. USENIX. URL: https://www.usenix.org/system/files/nsdi23-zhou.pdf.

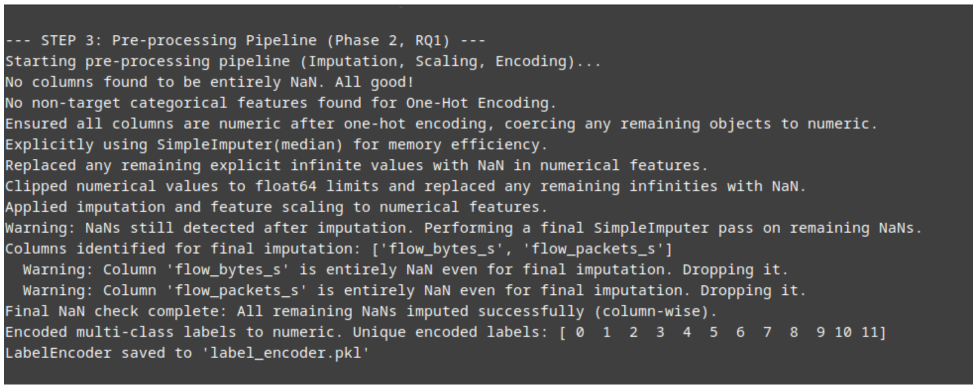
Zou, Deming, Kin Fun Lo, and Hongmei Kim (2021). “A survey of network intrusion detection methods based on machine learning”. In: *Journal of Network and Computer Applications* 190, p. 103132.

8 Appendix

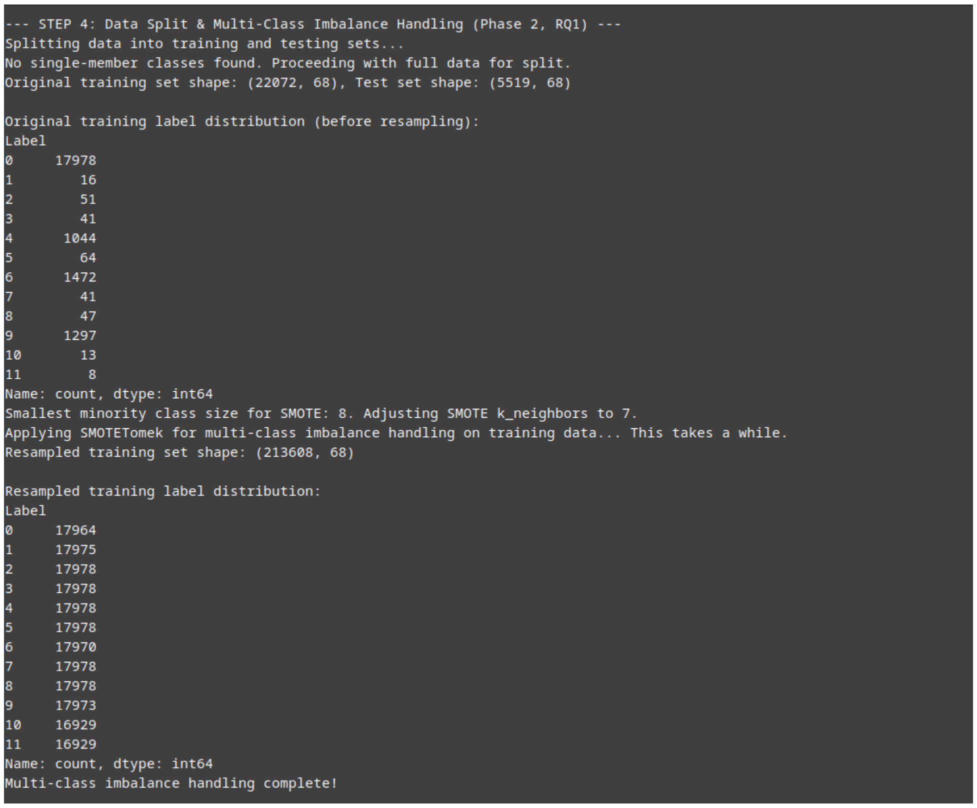
XGBoost on CIC-IDS2017 in NIDS pipeline outputs



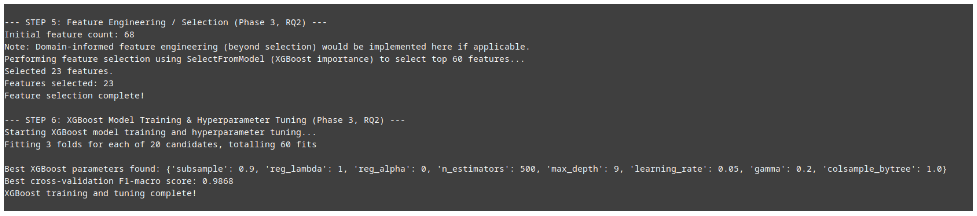
```
(venv) zeus@codec:~/Desktop/IDS$ python3 xgb_on_cicids2017_for_nids.py
Libraries imported successfully!
--- Initiating Pipeline ---

--- STEP 2: Loading & Initial Data Cleaning (Phase 2, RQ1) ---
Loading data from: MachineLearningCVE...
Sampling 1% of rows from each file for memory efficiency.
Processing file 1/8: Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv...
  Sampled 1704 rows from this file.
Processing file 2/8: Monday-WorkingHours.pcap_ISCX.csv...
  Sampled 5299 rows from this file.
Processing file 3/8: Wednesday-workingHours.pcap_ISCX.csv...
  Sampled 6927 rows from this file.
Processing file 4/8: Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv...
  Sampled 2886 rows from this file.
Processing file 5/8: Tuesday-WorkingHours.pcap_ISCX.csv...
  Sampled 4459 rows from this file.
Processing file 6/8: Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv...
  Sampled 2865 rows from this file.
Processing file 7/8: Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv...
  Sampled 2257 rows from this file.
Processing file 8/8: Friday-WorkingHours-Morning.pcap_ISCX.csv...
  Sampled 1910 rows from this file.
All files processed. Combined dataset size: 27591 rows and 79 columns.
Dropped constant or near-constant features from combined dataset: ['bwd_psh_flags', 'bwd_urg_flags', 'fwd_avg_bytes_bulk', 'fwd_avg_packets_bulk', 'fwd_avg_bulk_rate', 'bwd_avg_bytes_bulk', 'bwd_avg_packets_bulk
', 'bwd_avg_bulk_rate']. Shape: (27591, 71)
Final unique labels: ['BENIGN' 'WEB ATTACK �BRUTE FORCE' 'WEB ATTACK �XSS' 'DOS_HULK'
 'DOS_SLOWHTTPTEST' 'DOS_GOLDENEYE' 'DOS_SLOWLORIS' 'BRUTEFORCE_SSH'
 'BRUTEFORCE_FTP' 'PORTSCAN' 'DDOS' 'BOTNET']
```



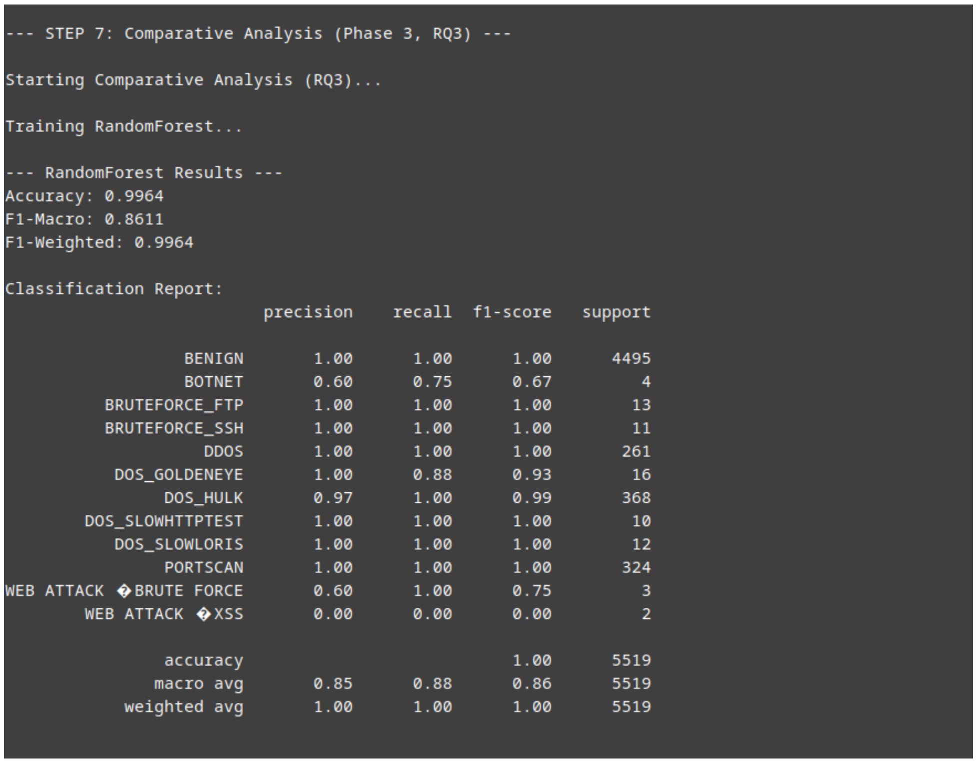
```
--- STEP 3: Pre-processing Pipeline (Phase 2, RQ1) ---
Starting pre-processing pipeline (Imputation, Scaling, Encoding)...
No columns found to be entirely NaN. All good!
No non-target categorical features found for One-Hot Encoding.
Ensured all columns are numeric after one-hot encoding, coercing any remaining objects to numeric.
Explicitly using SimpleImputer(median) for memory efficiency.
Replaced any remaining explicit infinite values with NaN in numerical features.
Clipped numerical values to float64 limits and replaced any remaining infinities with NaN.
Applied imputation and feature scaling to numerical features.
Warning: NaNs still detected after imputation. Performing a final SimpleImputer pass on remaining NaNs.
Columns identified for final imputation: ['flow_bytes_s', 'flow_packets_s']
  Warning: Column 'flow_bytes_s' is entirely NaN even for final imputation. Dropping it.
  Warning: Column 'flow_packets_s' is entirely NaN even for final imputation. Dropping it.
Final NaN check complete: All remaining NaNs imputed successfully (column-wise).
Encoded multi-class labels to numeric. Unique encoded labels: [ 0  1  2  3  4  5  6  7  8  9 10 11]
LabelEncoder saved to 'label_encoder.pkl'
```



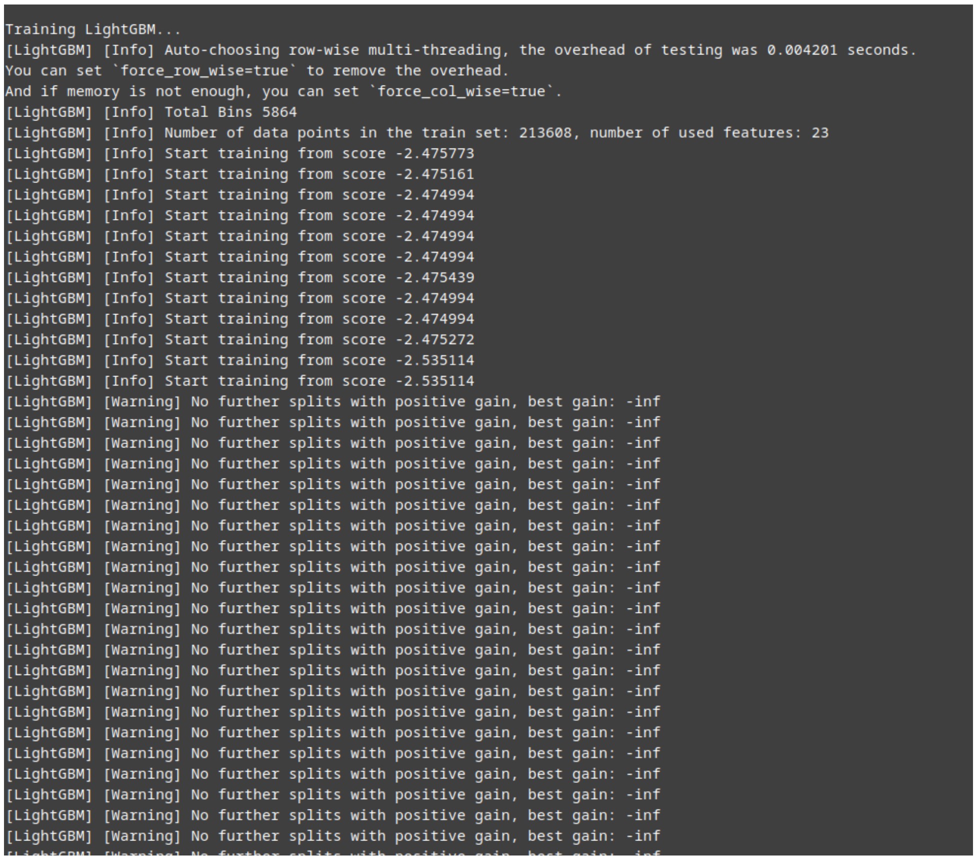
```
--- STEP 4: Data Split & Multi-Class Imbalance Handling (Phase 2, RQ1) ---
Splitting data into training and testing sets...
No single-member classes found. Proceeding with full data for split.
Original training set shape: (22072, 68), Test set shape: (5519, 68)

Original training label distribution (before resampling):
Label
0     17978
1        16
2        51
3        41
4      1044
5        64
6      1472
7        41
8        47
9      1297
10       13
11        8
Name: count, dtype: int64
Smallest minority class size for SMOTE: 8. Adjusting SMOTE k_neighbors to 7.
Applying SMOTETomek for multi-class imbalance handling on training data... This takes a while.
Resampled training set shape: (213608, 68)

Resampled training label distribution:
Label
0     17964
1     17975
2     17978
3     17978
4     17978
5     17978
6     17970
7     17978
8     17978
9     17973
10    16929
11    16929
Name: count, dtype: int64
Multi-class imbalance handling complete!
```



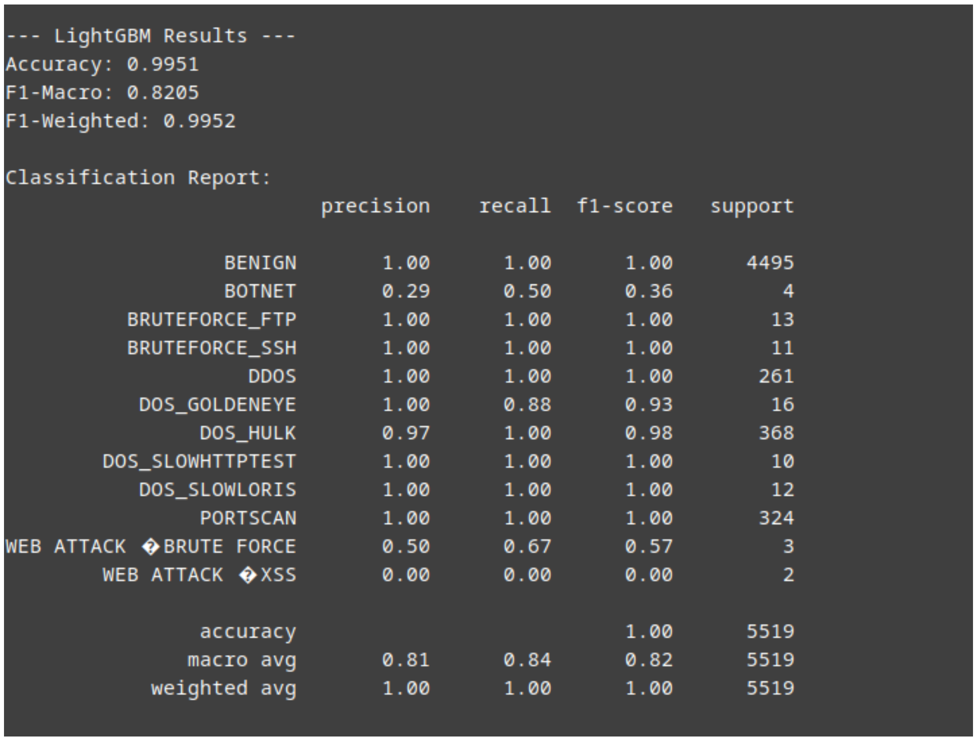
```
--- STEP 5: Feature Engineering / Selection (Phase 3, RQ2) ---
Initial feature count: 68
Note: Domain-informed feature engineering (beyond selection) would be implemented here if applicable.
Performing feature selection using SelectFromModel (XGBoost importance) to select top 60 features...
Selected 23 features.
Features selected: 23
Feature selection complete!

--- STEP 6: XGBoost Model Training & Hyperparameter Tuning (Phase 3, RQ2) ---
Starting XGBoost model training and hyperparameter tuning...
Fitting 3 folds for each of 20 candidates, totalling 60 fits

Best XGBoost parameters found: {'subsample': 0.9, 'reg_lambda': 1, 'reg_alpha': 0, 'n_estimators': 500, 'max_depth': 9, 'learning_rate': 0.05, 'gamma': 0.2, 'colsample_bytree': 1.0}
Best cross-validation F1-macro score: 0.9868
XGBoost training and tuning complete!
```



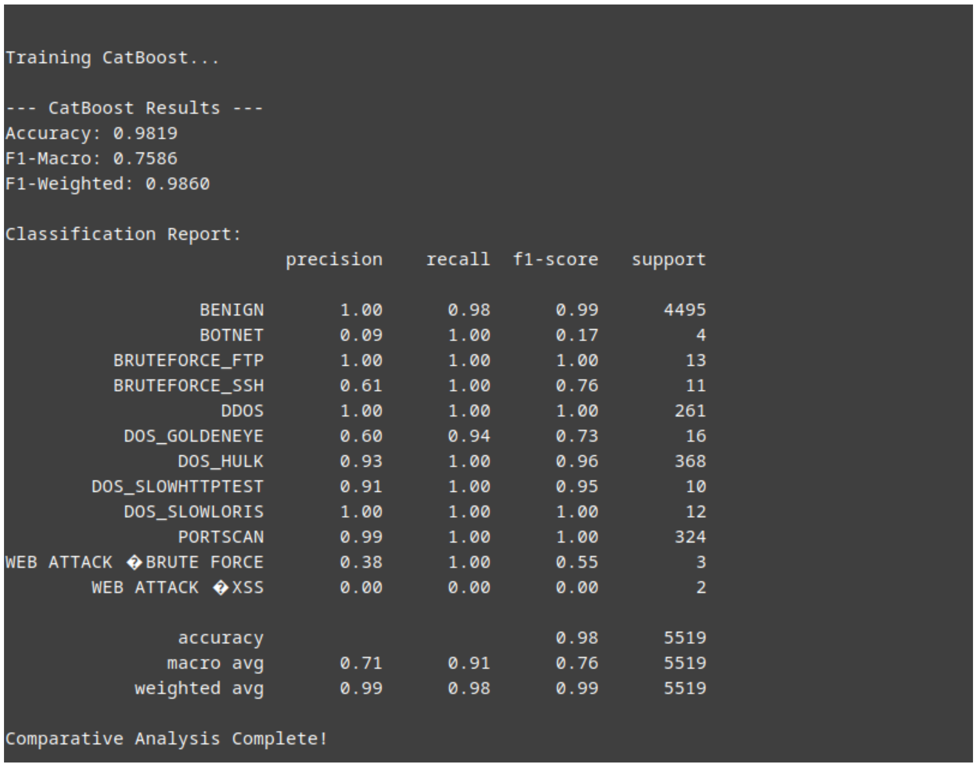
```
--- STEP 7: Comparative Analysis (Phase 3, RQ3) ---

Starting Comparative Analysis (RQ3)...

Training RandomForest...

--- RandomForest Results ---
Accuracy: 0.9964
F1-Macro: 0.8611
F1-Weighted: 0.9964

Classification Report:
                          precision    recall  f1-score   support

                  BENIGN       1.00      1.00      1.00      4495
                  BOTNET       0.60      0.75      0.67         4
          BRUTEFORCE_FTP       1.00      1.00      1.00        13
          BRUTEFORCE_SSH       1.00      1.00      1.00        11
                    DDOS       1.00      1.00      1.00       261
           DOS_GOLDENEYE       1.00      0.88      0.93        16
                DOS_HULK       0.97      1.00      0.99       368
        DOS_SLOWHTTPTEST       1.00      1.00      1.00        10
           DOS_SLOWLORIS       1.00      1.00      1.00        12
                PORTSCAN       1.00      1.00      1.00       324
WEB ATTACK �BRUTE FORCE       0.60      1.00      0.75         3
        WEB ATTACK �XSS       0.00      0.00      0.00         2

                accuracy                           1.00      5519
               macro avg       0.85      0.88      0.86      5519
            weighted avg       1.00      1.00      1.00      5519
```



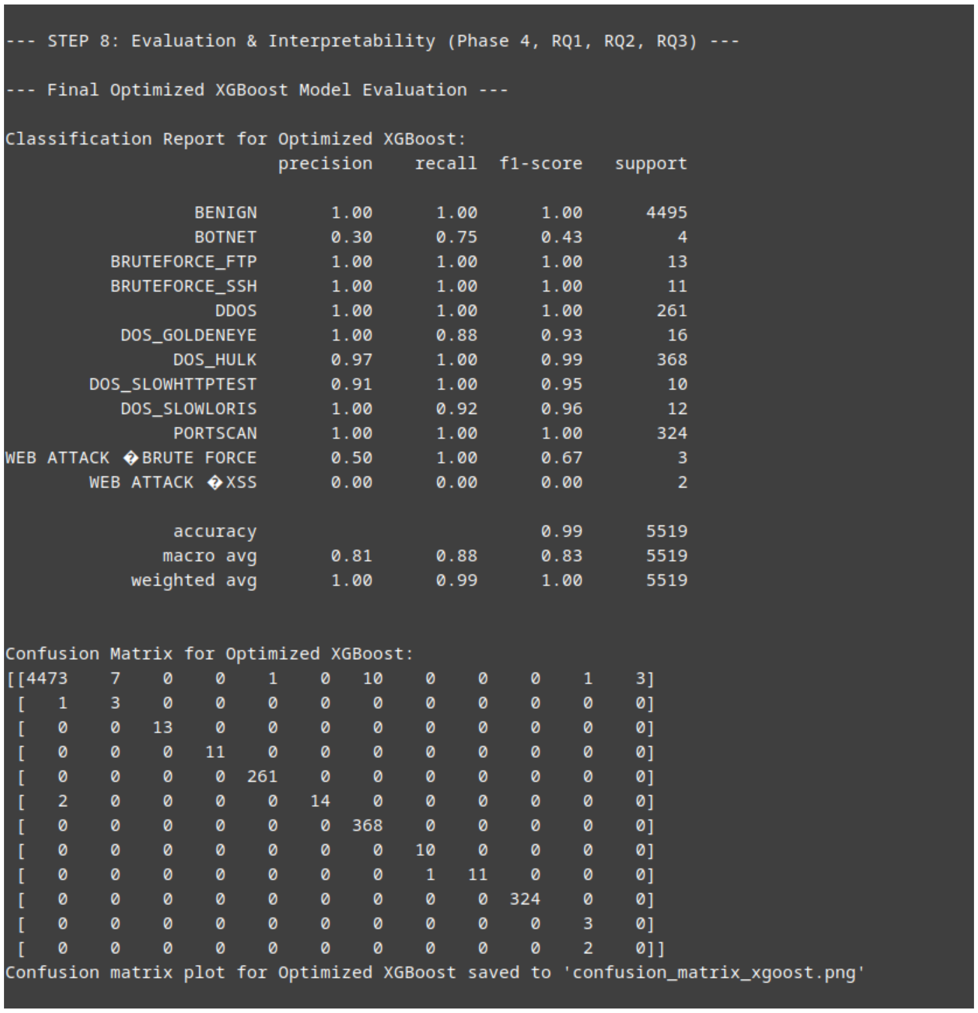
```
Training LightGBM...
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.004201 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 5864
[LightGBM] [Info] Number of data points in the train set: 213608, number of used features: 23
[LightGBM] [Info] Start training from score -2.475773
[LightGBM] [Info] Start training from score -2.475161
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.475439
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.474994
[LightGBM] [Info] Start training from score -2.475272
[LightGBM] [Info] Start training from score -2.535114
[LightGBM] [Info] Start training from score -2.535114
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```



```
--- LightGBM Results ---
Accuracy: 0.9951
F1-Macro: 0.8205
F1-Weighted: 0.9952

Classification Report:
                          precision    recall  f1-score   support

                BENIGN       1.00      1.00      1.00      4495
                BOTNET       0.29      0.50      0.36         4
        BRUTEFORCE_FTP       1.00      1.00      1.00        13
        BRUTEFORCE_SSH       1.00      1.00      1.00        11
                  DDOS       1.00      1.00      1.00       261
         DOS_GOLDENEYE       1.00      0.88      0.93        16
              DOS_HULK       0.97      1.00      0.98       368
      DOS_SLOWHTTPTEST       1.00      1.00      1.00        10
         DOS_SLOWLORIS       1.00      1.00      1.00        12
              PORTSCAN       1.00      1.00      1.00       324
WEB ATTACK �BRUTE FORCE       0.50      0.67      0.57         3
        WEB ATTACK �XSS       0.00      0.00      0.00         2

              accuracy                           1.00      5519
             macro avg       0.81      0.84      0.82      5519
          weighted avg       1.00      1.00      1.00      5519
```

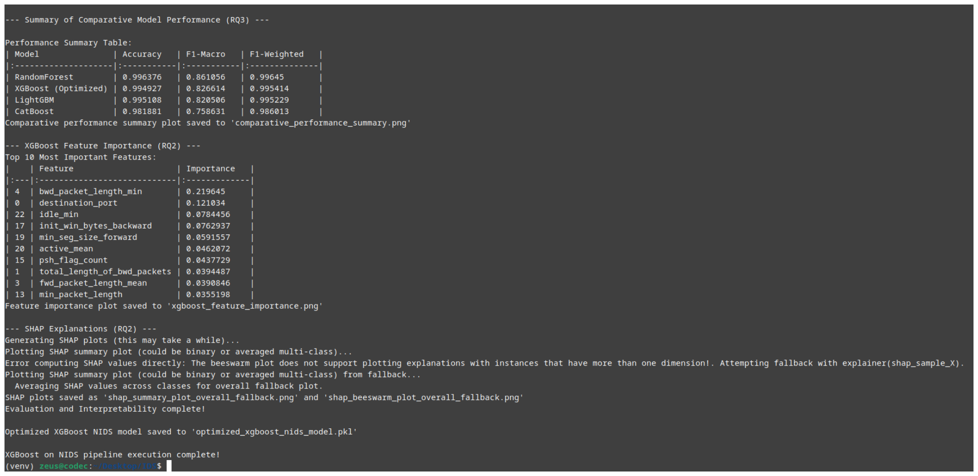


```
Training CatBoost...

--- CatBoost Results ---
Accuracy: 0.9819
F1-Macro: 0.7586
F1-Weighted: 0.9860

Classification Report:
                          precision    recall  f1-score   support

                BENIGN       1.00      0.98      0.99      4495
                BOTNET       0.09      1.00      0.17         4
        BRUTEFORCE_FTP       1.00      1.00      1.00        13
        BRUTEFORCE_SSH       0.61      1.00      0.76        11
                  DDOS       1.00      1.00      1.00       261
         DOS_GOLDENEYE       0.60      0.94      0.73        16
              DOS_HULK       0.93      1.00      0.96       368
      DOS_SLOWHTTPTEST       0.91      1.00      0.95        10
         DOS_SLOWLORIS       1.00      1.00      1.00        12
              PORTSCAN       0.99      1.00      1.00       324
WEB ATTACK �BRUTE FORCE       0.38      1.00      0.55         3
        WEB ATTACK �XSS       0.00      0.00      0.00         2

              accuracy                           0.98      5519
             macro avg       0.71      0.91      0.76      5519
          weighted avg       0.99      0.98      0.99      5519

Comparative Analysis Complete!
```



```
--- STEP 8: Evaluation & Interpretability (Phase 4, RQ1, RQ2, RQ3) ---

--- Final Optimized XGBoost Model Evaluation ---

Classification Report for Optimized XGBoost:
                          precision    recall  f1-score   support

                  BENIGN       1.00      1.00      1.00      4495
                  BOTNET       0.30      0.75      0.43         4
          BRUTEFORCE_FTP       1.00      1.00      1.00        13
          BRUTEFORCE_SSH       1.00      1.00      1.00        11
                    DDOS       1.00      1.00      1.00       261
           DOS_GOLDENEYE       1.00      0.88      0.93        16
                DOS_HULK       0.97      1.00      0.99       368
        DOS_SLOWHTTPTEST       0.91      1.00      0.95        10
           DOS_SLOWLORIS       1.00      0.92      0.96        12
                PORTSCAN       1.00      1.00      1.00       324
WEB ATTACK �BRUTE FORCE        0.50      1.00      0.67         3
        WEB ATTACK �XSS        0.00      0.00      0.00         2

                accuracy                           0.99      5519
               macro avg       0.81      0.88      0.83      5519
            weighted avg       1.00      0.99      1.00      5519


Confusion Matrix for Optimized XGBoost:
[[4473    7    0    0    1    0   10    0    0    0    1    3]
 [   1    3    0    0    0    0    0    0    0    0    0    0]
 [   0    0   13    0    0    0    0    0    0    0    0    0]
 [   0    0    0   11    0    0    0    0    0    0    0    0]
 [   0    0    0    0  261    0    0    0    0    0    0    0]
 [   2    0    0    0    0   14    0    0    0    0    0    0]
 [   0    0    0    0    0    0  368    0    0    0    0    0]
 [   0    0    0    0    0    0    0   10    0    0    0    0]
 [   0    0    0    0    0    0    0    1   11    0    0    0]
 [   0    0    0    0    0    0    0    0    0  324    0    0]
 [   0    0    0    0    0    0    0    0    0    0    3    0]
 [   0    0    0    0    0    0    0    0    0    0    2    0]]
Confusion matrix plot for Optimized XGBoost saved to 'confusion_matrix_xgoost.png'
```

%captionwhatweb output



```
--- Summary of Comparative Model Performance (RQ3) ---

Performance Summary Table:
| Model              | Accuracy   | F1-Macro   | F1-Weighted   |
|:-------------------|:-----------|:-----------|:--------------|
| RandomForest       | 0.996376   | 0.861056   | 0.99645       |
| XGBoost (Optimized) | 0.994927   | 0.826614   | 0.995414      |
| LightGBM           | 0.995108   | 0.820506   | 0.995229      |
| CatBoost           | 0.981881   | 0.758631   | 0.986013      |
Comparative performance summary plot saved to 'comparative_performance_summary.png'

--- XGBoost Feature Importance (RQ2) ---
Top 10 Most Important Features:
|    | Feature                     | Importance   |
|:---|:----------------------------|:-------------|
| 4  | bwd_packet_length_min       | 0.219645     |
| 0  | destination_port            | 0.121034     |
| 22 | idle_min                    | 0.0784456    |
| 17 | init_win_bytes_backward     | 0.0762937    |
| 19 | min_seg_size_forward        | 0.0591557    |
| 20 | active_mean                 | 0.0462072    |
| 15 | psh_flag_count              | 0.0437729    |
| 1  | total_length_of_bwd_packets | 0.0394487    |
| 3  | fwd_packet_length_mean      | 0.0390846    |
| 13 | min_packet_length           | 0.0355198    |
Feature importance plot saved to 'xgboost_feature_importance.png'

--- SHAP Explanations (RQ2) ---
Generating SHAP plots (this may take a while)...
Plotting SHAP summary plot (could be binary or averaged multi-class)...
Error computing SHAP values directly: The beeswarm plot does not support plotting explanations with instances that have more than one dimension!. Attempting fallback with explainer(shap_sample_X).
Plotting SHAP summary plot (could be binary or averaged multi-class) from fallback...
  Averaging SHAP values across classes for overall fallback plot.
SHAP plots saved as 'shap_summary_plot_overall_fallback.png' and 'shap_beeswarm_plot_overall_fallback.png'
Evaluation and Interpretability complete!

Optimized XGBoost NIDS model saved to 'optimized_xgboost_nids_model.pkl'

XGBoost on NIDS pipeline execution complete!
(venv) zeus@codec:~/Desktop/IDS$ 
```

Table 8: Dataset Features: Data Types, Null Values, Memory Usage, and Associated Commands

Feature	**Data Type**	**Null Values**	**Memory Usage (RAM)**	**Command**
DestinationPort	int64	0	581MB	.info()
Duration	float64	0	581MB	.info()
Flow_ID	int64	0	581MB	.info()
Label	object	2030 NaNs	2.9MB	.info()
Month	int64	0	581MB	.info()
New_Connections	int64	0	581MB	.info()
Packet_Direction	object	0	581MB	.info()
Packets_In_PerIOD	int64	0	581MB	.info()
Packets_Out_PerIOD	int64	0	581MB	.info()
Packet_Size	int64	0	581MB	.info()
Protocol	object	0	581MB	.info()
Source_Flags	int64	0	581MB	.info()
SourcePort	int64	0	581MB	.info()
Total_Flow_Packet_Count	int64	0	581MB	.info()
Total_Packet_Length	float64	0	581MB	.info()
Total_FlowBytes_Length	float64	0	581MB	.info()
Year	int64	0	581MB	.info()
FTP_ORIGINALFERETSIZE	int64	0	581MB	.info()
HTTPMethods	object	0	581MB	.info()
HTTPRequest_Referer	object	0	581MB	.info()
HTTPRequest_ResponseCode	object	0	581MB	.info()
HTTPRequest_UserAgent	object	0	581MB	.info()

Feature	**Data Type**	**Null Values**	**Memory Usage (RAM)**	**Command**
HTTPResponse_Code	object	0	581MB	.info()
JSON_JSON	object	0	581MB	.info()
Packet_Sequence_Number	int64	0	581MB	.info()
SFTP_OriginalFERETSIZE	int64	0	581MB	.info()
SSLVersion	object	0	581MB	.info()
SSHCommand_ID	int64	0	581MB	.info()
SSHVersion	object	0	581MB	.info()
TLSTicketLifetime	float64	0	581MB	.info()
FTPDataTransferSize	int64	0	581MB	.info()
FTPCommandID	int64	0	581MB	.info()
DNSQryName	object	0	581MB	.info()
DNSRR	object	0	581MB	.info()
Dst_Land	object	0	581MB	.info()
Dst_Lon	float64	0	581MB	.info()
Dst_Src_win_ratio	float64	0	581MB	.info()
Dst_Weird_flag_freq	float64	0	581MB	.info()
Dst_WinSize	float64	0	581MB	.info()
Dst_Mean_PacketSize	float64	0	581MB	.info()
Dst_Median_PacketSize	float64	0	581MB	.info()
Dst_Min_PacketSize	float64	0	581MB	.info()
Dst_Max_PacketSize	float64	0	581MB	.info()
Dst_Skew_PacketSize	float64	0	581MB	.info()
Dst_Src_connect_ratio	float64	0	581MB	.info()
Dst_Src_count_rate	float64	0	581MB	.info()
Dst_IP_packets_count	int64	0	581MB	.info()
Dst_Port_packets_count	int64	0	581MB	.info()
Dst_PortFrac	float64	0	581MB	.info()
Dst_PortZeroByteFraction	float64	0	581MB	.info()
Dst_PKTS	int64	0	581MB	.info()
Dst_TotalPacketLength	float64	0	581MB	.info()
Dst_IP_ZeroByteRate	float64	0	581MB	.info()
Dst_Port_ZeroByteRate	float64	0	581MB	.info()
Dst_TotalByteRate	float64	0	581MB	.info()
DstByteRateFrac	float64	0	581MB	.info()
DstUDPByteRateFrac	float64	0	581MB	.info()
Dst_ZeroByteFraction	float64	0	581MB	.info()
Dst_ZeroByteRate	float64	0	581MB	.info()

Feature	**Data Type**	**Null Values**	**Memory Usage (RAM)**	**Command**
DstBytesPerIPsec	float64	0	581MB	.info()
DstBytesPerPkts	float64	0	581MB	.info()
Dst_TCP_Mean_PacketSize	float64	0	581MB	.info()
Dst_TCP_Median_PacketSize	float64	0	581MB	.info()
Dst_TCP_Min_PacketSize	float64	0	581MB	.info()
Dst_TCP_Max_PacketSize	float64	0	581MB	.info()
Dst_TCP_Skew_PacketSize	float64	0	581MB	.info()
Dst_UDP_Mean_PacketSize	float64	0	581MB	.info()
Dst_UDP_Median_PacketSize	float64	0	581MB	.info()
Dst_UDP_Min_PacketSize	float64	0	581MB	.info()
Dst_UDP_Max_PacketSize	float64	0	581MB	.info()
Dst_UDP_Skew_PacketSize	float64	0	581MB	.info()
FailedLogins	int64	0	581MB	.info()
FlowBitsPerSec	float64	0	581MB	.info()
HTTPTransferSize	int64	6589 NaNs	581MB	.info()
Month_of_the_year	object	0	581MB	.info()
SFTP_RegularEXP_PERMSTATE	int64	0	581MB	.info()
SRC_IFACE	object	0	581MB	.info()
TotalDNS_Msg	int64	0	581MB	.info()
Src_Latitude	float64	0	581MB	.info()
Src_Latitude_ToInt	int64	0	581MB	.info()
Src_Lon	float64	0	581MB	.info()
Src_Lon_ToInt	int64	0	581MB	.info()
Src_Land	object	0	581MB	.info()
Src_Mean_PacketSize	float64	0	581MB	.info()
Src_Median_PacketSize	float64	0	581MB	.info()
Src_Min_PacketSize	float64	0	581MB	.info()
Src_Max_PacketSize	float64	0	581MB	.info()
Src_Skew_PacketSize	float64	0	581MB	.info()
Src_WinSize	float64	0	581MB	.info()
Src_IP_Address_Count	int64	0	581MB	.info()
Src_IP_packets_count	int64	0	581MB	.info()
Src_Port_packets_count	int64	0	581MB	.info()
Src_PortFrac	float64	0	581MB	.info()
Src_PKTS	int64	0	581MB	.info()
Src_TotalPacketLength	float64	0	581MB	.info()
Src_TotalByteRate	float64	0	581MB	.info()

Feature	**Data Type**	**Null Values**	**Memory Usage (RAM)**	**Command**
SrcByteRateFrac	float64	0	581MB	.info()
Src_TCP_Mean_PacketSize	float64	0	581MB	.info()
Src_TCP_Median_PacketSize	float64	0	581MB	.info()
Src_TCP_Min_PacketSize	float64	0	581MB	.info()
Src_TCP_Max_PacketSize	float64	0	581MB	.info()
Src_TCP_Skew_PacketSize	float64	0	581MB	.info()
Src_UDP_Mean_PacketSize	float64	0	581MB	.info()
Src_UDP_Median_PacketSize	float64	0	581MB	.info()
Src_UDP_Min_PacketSize	float64	0	581MB	.info()
Src_UDP_Max_PacketSize	float64	0	581MB	.info()
Src_UDP_Skew_PacketSize	float64	0	581MB	.info()
TimeHours	float64	0	581MB	.info()
Type_of_DDoS	object	0	581MB	.info()
UDP_MiscellaneousCommands	object	0	581MB	.info()
Weekdays	object	0	581MB	.info()
Is_Botnet_Attribute	object	0	581MB	.info()
Is_Infiltration_Attack	object	0	581MB	.info()
Is_DDOS_Attack	object	0	581MB	.info()
Is_DoS_Attack	object	0	581MB	.info()
Is_Heartbleed_Attack	object	0	581MB	.info()
Is_Web_Attack	object	0	581MB	.info()
Weekends	object	0	581MB	.info()
WeekHourGroups	object	0	581MB	.info()
Day_Occurences	int64	0	581MB	.info()
DaylightSavingsTriggerHours	object	0	581MB	.info()
DST_Code	int64	0	581MB	.info()
HoursOfDay	object	0	581MB	.info()
NonDaylightSavingTriggerHours	object	0	581MB	.info()
Number_Of_NonWeekDays	int64	0	581MB	.info()
SecondsInHour	int64	0	581MB	.info()
Weekday_Occurences	int64	0	581MB	.info()

Listing 1: Complete pipeline of XGBoost using CIC-IDS2017 in NIDS using Python

```python
# STEP 1: IMPORT LIBRARIES
import pandas as pd
import numpy as np
import glob
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler

# --- REQUIRED FOR IterativeImputer (as it's experimental) ---
from sklearn.experimental import enable_iterative_imputer
# ----------------------------------------------------------

from sklearn.impute import SimpleImputer, IterativeImputer  # For advanced
    imputation (RQ1)
from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score, f1_score, roc_auc_score, \
precision_recall_fscore_support
from sklearn.ensemble import RandomForestClassifier  # For comparative
    analysis (RQ3)
import xgboost as xgb  # For core model (RQ1, RQ2, RQ3)
import lightgbm as lgb  # For comparative analysis (RQ3)
import catboost as cb  # For comparative analysis (RQ3)
import shap  # For interpretability
from imblearn.over_sampling import SMOTE, ADASYN  # For multi-class
    oversampling strategies (RQ1)
from imblearn.combine import SMOTETomek  # For robust oversampling +
    undersampling (RQ1)
from imblearn.pipeline import Pipeline as ImbPipeline  # For robust pre-
    processing pipeline
from sklearn.feature_selection import SelectFromModel  # For feature
    selection (RQ2)
import joblib
import warnings
import gc  # For garbage collection

warnings.filterwarnings('ignore')  # Suppress warnings for cleaner output,
    but be cautious in real analysis

print("Libraries imported successfully!")


# ---
# Phase 2 & 3: Data Acquisition and Pre-processing & Model Development &
    Experimentation
# This section covers loading, initial cleaning, advanced pre-processing,
# data splitting, imbalance handling, feature selection, and model training
    /tuning.

# ---
# STEP 2: LOAD & INITIAL CLEAN DATA (Phase 2, aligns with RQ1)
# This step focuses on loading all CSVs and performing initial, critical
   cleaning
# to prepare for more advanced pre-processing strategies (aligns with RQ1).

def load_and_initial_clean_data(data_path, sampling_fraction=1.0,
   random_state=42):  # Added sampling_fraction
"""
Loads all CIC-IDS2017 CSV files from a specified path,
performs initial data cleaning (dropping irrelevant columns,
standardizing missing/infinite values, converting to numeric types),
and retains multi-class labels. This version is optimized for memory.
Includes an option to sample a fraction of the data from each file.
"""
print(f"\n--- STEP 2: Loading & Initial Data Cleaning (Phase 2, RQ1) ---")
print(f"Loading data from: {data_path}...")
if sampling_fraction < 1.0:
print(f"Sampling {sampling_fraction * 100:.0f}% of rows from each file for
   memory efficiency.")

file_paths = glob.glob(f'MachineLearningCVE/*.csv')  # Ensure this path is
   correct for your setup
if not file_paths:
raise ValueError(f"No CSV files found in {data_path}. Please check the path
   .")

full_df = pd.DataFrame()  # Initialize an empty DataFrame to append to

# Define common irrelevant columns and label variations
drop_cols = ['Flow ID', 'Source IP', 'Destination IP', 'Timestamp', '
   SimillarHTTP', 'Unnamed: 11']
label_replacements = {
        'DoS GoldenEye': 'DoS_GoldenEye', 'DoS Hulk': 'DoS_Hulk',
        'DoS Slowhttptest': 'DoS_Slowhttptest', 'DoS slowloris': '
           DoS_Slowloris',
        'Heartbleed': 'Heartbleed', 'Web Attack - Brute Force': '
           Web_Attack_Brute_Force',
        'Web Attack - XSS': 'Web_Attack_XSS', 'Web Attack - Sql Injection':
            'Web_Attack_SQL_Injection',
        'PortScan': 'PortScan', 'DDoS': 'DDoS', 'FTP-Patator': '
           BruteForce_FTP',
        'SSH-Patator': 'BruteForce_SSH', 'Infiltration': 'Infiltration', '
           Bot': 'Botnet',
        'NaN': 'NAN', ' nan': 'NAN', 'None': 'NAN', ' ': 'NAN'  #
           Standardize potential NaN strings
}

for i, f_path in enumerate(file_paths):
```

```
print(f"Processing file {i + 1}/{len(file_paths)}: {f_path.split('/')
    [-1]}...")
try:
# Load each CSV with low_memory=False to avoid mixed type warnings, but
    handle memory
temp_df = pd.read_csv(f_path, low_memory=False)

# --- Apply sampling fraction immediately after loading ---
if sampling_fraction < 1.0 and not temp_df.empty:
temp_df = temp_df.sample(frac=sampling_fraction, random_state=random_state)
print(f"  Sampled {temp_df.shape[0]} rows from this file.")
# --------------------------------------------------------

# Robust column name cleaning and standardization for current file
temp_df.columns = temp_df.columns.str.lower().str.strip().str.replace(' ',
    '_').str.replace('/',
'_').str.replace(
'-', '_').str.replace("'", "")

# Check for 'label' variations and rename to 'Label' (with uppercase L)
found_label_col = None
for col in temp_df.columns:
if 'label' in col:
found_label_col = col
break
if found_label_col and found_label_col != 'Label':
temp_df.rename(columns={found_label_col: 'Label'}, inplace=True)
elif not found_label_col and 'label' in temp_df.columns:  # Sometimes it
    might already be lowercase 'label'
temp_df.rename(columns={'label': 'Label'}, inplace=True)

if 'Label' not in temp_df.columns:
print(
f"Warning: 'Label' column not found in '{f_path.split('/')[-1]}'. Columns:
    {temp_df.columns.tolist()}")
# Optionally, skip this file if 'Label' is critical and missing
continue

# Drop irrelevant columns for this chunk
temp_df.drop(columns=[c for c in drop_cols if c in temp_df.columns],
    inplace=True, errors='ignore')

# Clean and standardize 'Label' column FIRST
temp_df['Label'] = temp_df['Label'].astype(str)
temp_df['Label'] = temp_df['Label'].replace(label_replacements).str.upper()
temp_df = temp_df[temp_df['Label'] != 'NAN']  # Drop rows where label is
    problematic
```

```
if temp_df.empty:
print(f"Skipping file '{f_path.split('/')[-1]}' as it became empty after
    label cleaning.")
del temp_df  # Free memory
gc.collect()
continue

# Process feature columns for this chunk
feature_columns = [col for col in temp_df.columns if col != 'Label']
for col in feature_columns:
if temp_df[col].dtype == 'object':
temp_df[col] = temp_df[col].replace(['Infinity', 'NaN', ' nan', 'None', ' '
    ], np.nan)
temp_df[col] = pd.to_numeric(temp_df[col], errors='coerce')

# Drop rows where all FEATURES are NaN for this chunk
initial_rows_before_feature_nan_drop = temp_df.shape[0]
temp_df.dropna(subset=feature_columns, how='all', inplace=True)
if temp_df.empty:
print(f"Skipping file '{f_path.split('/')[-1]}' as it became empty after
    feature NaN drop.")
del temp_df
gc.collect()
continue

# Remove duplicates for this chunk
temp_df.drop_duplicates(inplace=True)

# Append to the full DataFrame
full_df = pd.concat([full_df, temp_df], ignore_index=True)

del temp_df  # Free memory
gc.collect()  # Trigger garbage collection

except Exception as e:
print(f"Error processing {f_path}: {e}")
continue

print(f"All files processed. Combined dataset size: {full_df.shape[0]} rows
     and {full_df.shape[1]} columns.")

if full_df.empty:
raise ValueError(
"All dataframes were processed, but the combined dataset is empty. Check
    individual files or cleaning steps.")

# Final cleaning after all files are combined
# Drop constant features from the full_df (excluding 'Label')
```

```
features_to_check_for_constancy = [col for col in full_df.columns if col !=
     'Label']
constant_features_found = [col for col in features_to_check_for_constancy
    if full_df[col].nunique() <= 1]

if constant_features_found:
full_df.drop(columns=constant_features_found, inplace=True, errors='ignore'
    )
print(
f"Dropped constant or near-constant features from combined dataset: {
    constant_features_found}. Shape: {full_df.shape}")
else:
print("No constant or near-constant features found in combined dataset (
    excluding 'Label').")

print(f"Final unique labels: {full_df['Label'].unique()}")

return full_df


# ---
# STEP 3: PRE-PROCESSING PIPELINE (Imputation, Scaling, Encoding) (Phase 2,
     aligns with RQ1)
# This step builds a robust pipeline for handling missing values and
    scaling numerical features.
# It also includes Label Encoding for the multi-class target variable.

def preprocess_data(df, random_state=42):
"""
Applies imputation, scaling, and encodes categorical features.
Retains multi-class labels for the target variable.
"""
print("\n--- STEP 3: Pre-processing Pipeline (Phase 2, RQ1) ---")
print("Starting pre-processing pipeline (Imputation, Scaling, Encoding)..."
    )

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

# --- NEW: Drop columns that are entirely NaN *before* any imputation ---
initial_feature_count = X.shape[1]
cols_to_drop_all_nan = X.columns[X.isnull().all()].tolist()
if cols_to_drop_all_nan:
X.drop(columns=cols_to_drop_all_nan, inplace=True)
print(
f"Dropped {len(cols_to_drop_all_nan)} columns that were entirely NaN: {
    cols_to_drop_all_nan}. Remaining features: {X.shape[1]}")
else:
```

```
print("No columns found to be entirely NaN. All good!")
# --- END NEW ---

# --- Handle Categorical Features (Non-Target) ---
# Identify non-numeric columns remaining after initial cleaning
categorical_cols = X.select_dtypes(include=['object', 'category']).columns
if not categorical_cols.empty:
# One-Hot Encoding for categorical features
X = pd.get_dummies(X, columns=categorical_cols, prefix=categorical_cols,
   drop_first=True)
print(f"One-Hot Encoded categorical features: {list(categorical_cols)}. New
    feature count: {X.shape[1]}")
else:
print("No non-target categorical features found for One-Hot Encoding.")

# --- NEW: Ensure all columns are truly numeric after all conversions ---
# This catches any remaining 'object' types that might contain NaNs or
   mixed data
for col in X.columns:
if X[col].dtype == 'object':
X[col] = pd.to_numeric(X[col], errors='coerce')
print("Ensured all columns are numeric after one-hot encoding, coercing any
    remaining objects to numeric.")
# --- END NEW ---

# --- Imputation and Scaling Pipeline ---
use_iterative_imputer = False  # Using SimpleImputer for memory efficiency

if use_iterative_imputer:
try:
imputer = IterativeImputer(max_iter=5, random_state=random_state, verbose
   =0)
print("Using IterativeImputer for advanced missing value imputation.")
except ImportError:
print(
"IterativeImputer not available (requires scikit-learn >= 0.23). Falling
   back to SimpleImputer(median).")
imputer = SimpleImputer(strategy='median')
else:
print("Explicitly using SimpleImputer(median) for memory efficiency.")
imputer = SimpleImputer(strategy='median')

scaler = StandardScaler()  # Feature scaling (RQ1)

# Create a pre-processing pipeline for numerical features
numeric_features_pipeline = ImbPipeline([
('imputer', imputer),
('scaler', scaler)
])

# Fit and transform numerical features
# Re-select numeric_cols as they might have changed after get_dummies and
    final to_numeric
current_numeric_cols = X.select_dtypes(include=np.number).columns

if not current_numeric_cols.empty:
# Ensure no infinities remain before imputation pipeline
X[current_numeric_cols].replace([np.inf, -np.inf], np.nan, inplace=True)
print("Replaced any remaining explicit infinite values with NaN in
    numerical features.")

# Safeguard against values too large/small for float64 before imputation
max_float = np.finfo(np.float64).max
min_float = np.finfo(np.float64).min
# Use .clip to cap values within float64 range, then re-replace any
    potential new infinities (e.g., from operations)
X[current_numeric_cols] = X[current_numeric_cols].clip(lower=min_float,
    upper=max_float).replace(
[np.inf, -np.inf], np.nan)
print("Clipped numerical values to float64 limits and replaced any
    remaining infinities with NaN.")

# Now, apply imputation and scaling to the cleaned numerical columns
X_transformed_numerical = numeric_features_pipeline.fit_transform(X[
    current_numeric_cols])
X[current_numeric_cols] = pd.DataFrame(X_transformed_numerical, columns=
    current_numeric_cols, index=X.index)
print("Applied imputation and feature scaling to numerical features.")
else:
print("No numerical features found for imputation and scaling after all
    processing.")

# --- FINAL NaN CHECK: Ensure absolutely no NaNs are left across *all*
    columns ---
if X.isnull().sum().sum() > 0:
print("Warning: NaNs still detected after imputation. Performing a final
    SimpleImputer pass on remaining NaNs.")

cols_with_nans_final = X.columns[X.isnull().any()].tolist()

if cols_with_nans_final:
print(f"Columns identified for final imputation: {cols_with_nans_final}")

for col_name in cols_with_nans_final:
if col_name in X.columns and X[col_name].isnull().any():  # Double check it
     still has NaNs and exists
```

```
median_val = X[col_name].median()
if pd.isna(median_val):  # If median is NaN, it means the entire column is
    NaN.
print(f"  Warning: Column '{col_name}' is entirely NaN even for final
    imputation. Dropping it.")
X.drop(columns=[col_name], inplace=True)
else:
X[col_name].fillna(median_val, inplace=True)
print(f"  Imputed NaNs in column '{col_name}' with median: {median_val}")
elif col_name not in X.columns:
print(f"  Warning: Column '{col_name}' identified for imputation but not
    found in X. Skipping.")

# Final check after individual column imputation
if X.isnull().sum().sum() > 0:
print(
f"Final check found {X.isnull().sum().sum()} NaNs remaining after column-
    wise imputation. This is unexpected.")
else:
print("Final NaN check complete: All remaining NaNs imputed successfully (
    column-wise).")

else:
print(
"No columns actually needed final NaN imputation despite overall NaN count
    being > 0. All NaNs effectively removed.")
else:
print("No NaNs detected after imputation. Data is clean!")

# --- Encode Target Label ---
le = LabelEncoder()
y_encoded = le.fit_transform(y)
print(f"Encoded multi-class labels to numeric. Unique encoded labels: {np.
    unique(y_encoded)}")
# Store the LabelEncoder for inverse transformation later
joblib.dump(le, 'label_encoder.pkl')
print("LabelEncoder saved to 'label_encoder.pkl'")

return X, pd.Series(y_encoded, name='Label'), le


# ---
# STEP 4: DATA SPLIT AND MULTI-CLASS IMBALANCE HANDLING (Phase 2, aligns
    with RQ1)
# This step performs the train-test split and applies advanced multi-class
    imbalance handling.
```

```
def split_and_handle_imbalance(X, y_encoded, test_size=0.2, random_state
    =42):
"""
Splits data into training and testing sets, then applies SMOTETomek
for robust multi-class imbalance handling on the training set.
"""
print("\n--- STEP 4: Data Split & Multi-Class Imbalance Handling (Phase 2,
    RQ1) ---")
print("Splitting data into training and testing sets...")


class_counts = pd.Series(y_encoded).value_counts()
single_member_classes = class_counts[class_counts == 1].index.tolist()

if single_member_classes:
print(f"Warning: Found classes with only 1 member (indices: {
    single_member_classes}). "
f"These samples will be removed before train_test_split to enable
    stratification.")

# Get the original labels for these single-member classes (for better
    logging)
le = joblib.load('label_encoder.pkl')  # Load the encoder to inverse
    transform
original_labels = le.inverse_transform(single_member_classes)
print(f"  Corresponding original labels: {original_labels.tolist()}")

# Filter out rows corresponding to these single-member classes
indices_to_keep = ~y_encoded.isin(single_member_classes)
X_filtered = X[indices_to_keep]
y_encoded_filtered = y_encoded[indices_to_keep]

print(f"  Removed {len(y_encoded) - len(y_encoded_filtered)} samples from
    single-member classes.")
print(f"  Remaining data shape for split: X={X_filtered.shape}, y={
    y_encoded_filtered.shape}")

X_to_split = X_filtered
y_to_split = y_encoded_filtered
else:
X_to_split = X
y_to_split = y_encoded
print("No single-member classes found. Proceeding with full data for split.
    ")


X_train, X_test, y_train, y_test = train_test_split(
X_to_split, y_to_split, test_size=test_size, random_state=random_state,
   stratify=y_to_split
)
print(f"Original training set shape: {X_train.shape}, Test set shape: {
   X_test.shape}")

# Display original training label distribution BEFORE resampling
print("\nOriginal training label distribution (before resampling):")
original_train_counts = pd.Series(y_train).value_counts().sort_index()
print(original_train_counts)

# --- NEW: Dynamically determine k_neighbors for SMOTE ---
# Identify minority classes (those not the majority)
class_counts = pd.Series(y_train).value_counts()
majority_class_label = class_counts.idxmax()
minority_class_counts = class_counts[class_counts.index !=
   majority_class_label]

if minority_class_counts.empty:
print("No minority classes found for oversampling. Skipping SMOTETomek.")
return X_train, X_test, y_train, y_test

min_minority_class_size = minority_class_counts.min()

# SMOTE(k_neighbors=K) requires at least K+1 samples in a class.
# So, K must be <= (min_minority_class_size - 1).
# We also need K to be at least 1.
smote_k_neighbors = max(1, min_minority_class_size - 1)

if smote_k_neighbors == 0:
print(
"Warning: Smallest minority class has only 1 sample. SMOTE cannot be
   applied to this class. Setting k_neighbors to 1, but it might not be
   oversampled.")
smote_k_neighbors = 1  # Smallest valid k_neighbors

print(
f"Smallest minority class size for SMOTE: {min_minority_class_size}.
   Adjusting SMOTE k_neighbors to {smote_k_neighbors}.")
# --- END NEW ---

# Use SMOTETomek for robust multi-class oversampling and cleaning (RQ1)
# 'auto' strategy balances all minority classes to the majority class.
# Adjusting `smote` and `tomek` parameters could be part of RQ1's "
   optimized resampling"
print("Applying SMOTETomek for multi-class imbalance handling on training
   data... This takes a while.")
smotetomek = SMOTETomek(
random_state=random_state,
sampling_strategy='auto',
smote=SMOTE(random_state=random_state, k_neighbors=smote_k_neighbors),  #
   Using dynamic k_neighbors
tomek=None
)
X_train_res, y_train_res = smotetomek.fit_resample(X_train, y_train)

print(f"Resampled training set shape: {X_train_res.shape}")
print("\nResampled training label distribution:")
resampled_train_counts = pd.Series(y_train_res).value_counts().sort_index()
print(resampled_train_counts)
print("Multi-class imbalance handling complete!")

return X_train_res, X_test, y_train_res, y_test


# ---
# STEP 5: FEATURE ENGINEERING / SELECTION (Phase 3, aligns with RQ2)
# This step applies feature selection using XGBoost's feature importances.
# A placeholder for more advanced domain-informed feature engineering is
   also included.

class FeatureEngineer(object):
"""
A placeholder class for more advanced, domain-informed feature engineering
and for performing feature selection using a model's feature importances.
"""

def __init__(self, n_features_to_select=None, random_state=42):
self.n_features_to_select = n_features_to_select
self.selector = None
self.feature_names_in_ = None
self.random_state = random_state

def fit(self, X, y):
self.feature_names_in_ = X.columns.tolist()
if self.n_features_to_select is not None and self.n_features_to_select < X.
   shape[1]:
print(
f"Performing feature selection using SelectFromModel (XGBoost importance)
   to select top {self.n_features_to_select} features...")
# Using a simple XGBoost model to get feature importances for selection
# eval_metric='mlogloss' is more robust for multi-class
model_for_selection = xgb.XGBClassifier(
random_state=self.random_state,
n_estimators=100,
learning_rate=0.1,
use_label_encoder=False,
eval_metric='mlogloss',
objective='multi:softmax',
num_class=len(np.unique(y))
)
model_for_selection.fit(X, y)
self.selector = SelectFromModel(model_for_selection, max_features=self.
   n_features_to_select, prefit=True)
# Fit the selector to the model_for_selection (which is already fitted)
# We need to explicitly fit SelectFromModel on X,y so it gets
   feature_importances_ and thresholds correctly
self.selector.fit(X, y)
# FIX: Changed .n_features_ to .get_support().sum()
print(f"Selected {self.selector.get_support().sum()} features.")
else:
print(
f"Skipping SelectFromModel as n_features_to_select ({self.
   n_features_to_select}) is not specified or is >= initial feature count
   ({X.shape[1]}).")
return self

def transform(self, X):
if self.selector:
# Get the names of the selected features
selected_feature_names = X.columns[self.selector.get_support()]
return pd.DataFrame(self.selector.transform(X), columns=
   selected_feature_names, index=X.index)
return X

def get_feature_names_out(self, input_features=None):
if self.selector:
# Ensure input_features is not None and is indexable
if input_features is not None and hasattr(input_features, '__getitem__'):
return input_features[self.selector.get_support()]
else:
# Fallback if input_features is not suitable, use internal names
return np.array(self.feature_names_in_)[self.selector.get_support()]
return input_features if input_features is not None else self.
   feature_names_in_


def perform_feature_selection(X_train_res, X_test, y_train_res, y_test,
   num_features=50, random_state=42):
"""
Applies feature selection based on XGBoost feature importances (RQ2).
"""
print(f"\n--- STEP 5: Feature Engineering / Selection (Phase 3, RQ2) ---")
print(f"Initial feature count: {X_train_res.shape[1]}")
```

```
# Placeholder for advanced domain-informed feature engineering (RQ2)
# Example: X_train_res = add_temporal_features(X_train_res)
# Example: X_test = add_temporal_features(X_test)
print("Note: Domain-informed feature engineering (beyond selection) would
   be implemented here if applicable.")

fe = FeatureEngineer(n_features_to_select=num_features, random_state=
   random_state)
fe.fit(X_train_res, y_train_res)  # Fit feature selector only on training
   data

X_train_selected = fe.transform(X_train_res)
X_test_selected = fe.transform(X_test)

print(f"Features selected: {X_train_selected.shape[1]}")
print("Feature selection complete!")

return X_train_selected, X_test_selected


# ---
# STEP 6: XGBOOST MODEL TRAINING & HYPERPARAMETER TUNING (Phase 3, aligns
   with RQ2)
# This step involves training the optimized XGBoost model with
   hyperparameter tuning.

def train_and_tune_xgboost(X_train_selected, y_train_res, le, random_state
   =42):
"""
Trains and tunes an XGBoost classifier using RandomizedSearchCV (RQ2).
"""
print("\n--- STEP 6: XGBoost Model Training & Hyperparameter Tuning (Phase
   3, RQ2) ---")
print("Starting XGBoost model training and hyperparameter tuning...")

# Define XGBoost model (multi-class)
# use_label_encoder=False is important for newer versions of XGBoost
# eval_metric='mlogloss' is suitable for multi-class classification
# objective='multi:softmax' for class labels, 'multi:softprob' for
   probabilities
model_xgb = xgb.XGBClassifier(
objective='multi:softmax',
num_class=len(le.classes_),  # Number of unique classes in target
eval_metric='mlogloss',
use_label_encoder=False,
n_jobs=-1,  # Use all available cores
random_state=random_state
)

# Define hyperparameters for RandomizedSearchCV (RQ2)
# A wide range for exploration, more granular tuning can follow with
   GridSearchCV
param_dist = {
        'n_estimators': [100, 200, 300, 500],  # Reduced for speed, can
           increase for thoroughness
        'learning_rate': [0.01, 0.05, 0.1],  # Reduced for speed
        'max_depth': [3, 5, 7, 9],  # Reduced for speed
        'subsample': [0.7, 0.8, 0.9, 1.0],
        'colsample_bytree': [0.7, 0.8, 0.9, 1.0],
        'gamma': [0, 0.1, 0.2],
        'reg_alpha': [0, 0.001, 0.01],  # L1 regularization (RQ2)
        'reg_lambda': [1, 10],  # L2 regularization (RQ2)
        # For multi-class, SMOTETomek handles the balancing.
        # If explicit class weights were needed, they'd be passed to `
           sample_weight` in .fit()
}

# Randomized search for hyperparameter tuning (RQ2)
# n_iter controls the number of parameter settings that are sampled.
# cv=3 or 5 for cross-validation folds.
# scoring='f1_macro' is good for imbalanced multi-class (RQ1, RQ2).
random_search = RandomizedSearchCV(
estimator=model_xgb,
param_distributions=param_dist,
n_iter=20,  # Reduced for speed, increase for more thorough search (e.g.,
   50-100)
scoring='f1_macro',  # Optimise for macro F1-score (RQ1, RQ2)
cv=3,  # 3-fold cross-validation (can increase to 5 for more robust results
   )
verbose=1,
random_state=random_state,
n_jobs=-1  # Use all available cores for parallel search
)

random_search.fit(X_train_selected, y_train_res)

best_xgb_model = random_search.best_estimator_
print(f"\nBest XGBoost parameters found: {random_search.best_params_}")
print(f"Best cross-validation F1-macro score: {random_search.best_score_:.4
   f}")
print("XGBoost training and tuning complete!")

return best_xgb_model
```

```
# ---
# STEP 7: COMPARATIVE ANALYSIS (Phase 3, aligns with RQ3)
# This step trains and evaluates other state-of-the-art models for
   comparison.

def perform_comparative_analysis(X_train_selected, X_test_selected,
   y_train_res, y_test, le, random_state=42):
"""
Trains and evaluates other state-of-the-art models for comparative analysis
    (RQ3).
"""
print("\n--- STEP 7: Comparative Analysis (Phase 3, RQ3) ---")
print("\nStarting Comparative Analysis (RQ3)...")

models = {
        'RandomForest': RandomForestClassifier(random_state=random_state,
           n_estimators=100, n_jobs=-1),
        'LightGBM': lgb.LGBMClassifier(
        objective='multiclass',
        num_class=len(le.classes_),
        random_state=random_state,
        n_estimators=300,  # Example: tuned, can be more extensively tuned
        learning_rate=0.05,
        n_jobs=-1
        ),
        'CatBoost': cb.CatBoostClassifier(
        objective='MultiClass',
        classes_count=len(le.classes_),
        random_state=random_state,
        iterations=300,  # Example: tuned
        learning_rate=0.05,
        verbose=0,  # Suppress verbose output during training
        thread_count=-1  # Use all available cores
        )
}

results = {}
for name, model in models.items():
print(f"\nTraining {name}...")
model.fit(X_train_selected, y_train_res)
y_pred = model.predict(X_test_selected)

# Calculate F1-score for multi-class
f1_macro = f1_score(y_test, y_pred, average='macro', zero_division=0)
f1_weighted = f1_score(y_test, y_pred, average='weighted', zero_division=0)
accuracy = accuracy_score(y_test, y_pred)

print(f"\n--- {name} Results ---")
print(f"Accuracy: {accuracy:.4f}")
print(f"F1-Macro: {f1_macro:.4f}")
print(f"F1-Weighted: {f1_weighted:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_,
zero_division=0))  # zero_division=0 to handle classes with no predicted
   samples

results[name] = {
        'accuracy': accuracy,
        'f1_macro': f1_macro,
        'f1_weighted': f1_weighted,
        'report': classification_report(y_test, y_pred, target_names=le.
           classes_, output_dict=True,
        zero_division=0),
        'confusion_matrix': confusion_matrix(y_test, y_pred)
}
print("Comparative Analysis Complete!")
return results


# ---
# Phase 4: Result Analysis
# This section focuses on evaluating models, visualizing results, and
   providing interpretability.

# ---
# STEP 8: EVALUATION AND INTERPRETABILITY (Phase 4, aligns with RQ1, RQ2,
   RQ3)
# This step evaluates the best XGBoost model and summarizes comparative
   results,
# and provides interpretability insights (RQ1, RQ2, RQ3).
def evaluate_and_interpret(best_xgb_model, comparative_results,
   X_test_selected, y_test, le, random_state=42):
"""
Evaluates the best XGBoost model, summarizes comparative results,
and provides interpretability insights (RQ1, RQ2, RQ3).
"""
print("\n--- STEP 8: Evaluation & Interpretability (Phase 4, RQ1, RQ2, RQ3)
    ---")

# --- Final Optimized XGBoost Model Evaluation (RQ1, RQ2) ---
print("\n--- Final Optimized XGBoost Model Evaluation ---")
y_pred_xgb = best_xgb_model.predict(X_test_selected)

# Classification Report
print("\nClassification Report for Optimized XGBoost:")
print(classification_report(y_test, y_pred_xgb, target_names=le.classes_,
    zero_division=0))

# Confusion Matrix
cm_xgb = confusion_matrix(y_test, y_pred_xgb)
print("\nConfusion Matrix for Optimized XGBoost:")
print(cm_xgb)

# Plotting Confusion Matrix
plt.figure(figsize=(12, 10))
sns.heatmap(cm_xgb, annot=True, fmt='d', cmap='Blues', xticklabels=le.
    classes_, yticklabels=le.classes_)
plt.title('Confusion Matrix for Optimized XGBoost (Test Set)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.savefig('confusion_matrix_xgboost.png')
plt.show()
print("Confusion matrix plot for Optimized XGBoost saved to '
    confusion_matrix_xgoost.png'")

# --- Summarize Comparative Analysis Results (RQ3) ---
print("\n--- Summary of Comparative Model Performance (RQ3) ---")
summary_data = []
# Add XGBoost results to summary
f1_xgb_macro = f1_score(y_test, y_pred_xgb, average='macro', zero_division
    =0)
f1_xgb_weighted = f1_score(y_test, y_pred_xgb, average='weighted',
    zero_division=0)
acc_xgb = accuracy_score(y_test, y_pred_xgb)
summary_data.append(
{'Model': 'XGBoost (Optimized)', 'Accuracy': acc_xgb, 'F1-Macro':
    f1_xgb_macro, 'F1-Weighted': f1_xgb_weighted})

for name, res in comparative_results.items():
summary_data.append({'Model': name, 'Accuracy': res['accuracy'], 'F1-Macro'
    : res['f1_macro'],
          'F1-Weighted': res['f1_weighted']})

summary_df = pd.DataFrame(summary_data).set_index('Model').sort_values(by='
    F1-Macro', ascending=False)
print("\nPerformance Summary Table:")
print(summary_df.to_markdown(numalign="left", stralign="left"))  # Nicely
    formatted table

# Plotting Summary
summary_df.plot(kind='bar', figsize=(14, 7))
plt.title('Comparative Model Performance (F1-Macro on Test Set)')
plt.ylabel('Score')
plt.xticks(rotation=45, ha='right')
plt.legend(loc='lower right')
plt.tight_layout()
plt.savefig('comparative_performance_summary.png')
plt.show()
print("Comparative performance summary plot saved to '
    comparative_performance_summary.png'")

# --- Feature Importance (RQ2) ---
print("\n--- XGBoost Feature Importance (RQ2) ---")
importance_df = pd.DataFrame({
        'Feature': X_test_selected.columns,
        'Importance': best_xgb_model.feature_importances_
}).sort_values(by='Importance', ascending=False)
print("Top 10 Most Important Features:")
print(importance_df.head(10).to_markdown(numalign="left", stralign="left"))

plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=importance_df.head(20))
plt.title('Top 20 XGBoost Feature Importances (Optimized Model)')
plt.tight_layout()
plt.savefig('xgboost_feature_importance.png')
plt.show()
print("Feature importance plot saved to 'xgboost_feature_importance.png'")

# --- SHAP for Interpretability (RQ2) ---
print("\n--- SHAP Explanations (RQ2) ---")
# Using a subset of the test data for SHAP for computational efficiency
# Adjust as needed, but for multi-class SHAP summary plots, it's often more
     practical
# to use a smaller, representative subset of the data for explainer
    computation.
print("Generating SHAP plots (this may take a while)...")
shap_sample_X = X_test_selected.sample(n=min(1000, len(X_test_selected)),
    random_state=random_state)

# Explain all classes for multi-class classification
explainer = shap.TreeExplainer(best_xgb_model)

# Determine expected_value for overall explanations (scalar)
if isinstance(explainer.expected_value, (list, np.ndarray)):
overall_base_value = explainer.expected_value[0]  # Take the first for
    overall/binary
else:
overall_base_value = explainer.expected_value  # Already scalar

try:
```

```
# This is the standard way to get SHAP values for multi-class tree models
shap_values_raw = explainer.shap_values(shap_sample_X)

if isinstance(shap_values_raw, list) and len(shap_values_raw) > 1:  # Multi
    -class output
# This path is already handled correctly for individual classes

num_classes_to_plot = min(3, len(le.classes_))
print(f"Multi-class SHAP values detected. Plotting for {num_classes_to_plot
    } key classes...")

expected_values_list = explainer.expected_value  # This should be an array/
    list for multi-class
if not isinstance(expected_values_list, (list, np.ndarray)):
expected_values_list = [expected_values_list] * len(le.classes_)

for i in range(num_classes_to_plot):
class_name = le.inverse_transform([i])[0]
if class_name == 'BENIGN' and len(le.classes_) > 1:
print(f"Skipping SHAP plot for BENIGN class (Index {i}).")
continue

print(f"Plotting SHAP for class: '{class_name}' (Index {i})")

explanation_for_class = shap.Explanation(
values=shap_values_raw[i],
base_values=expected_values_list[i],
data=shap_sample_X.values,  # Pass as NumPy array here too for consistency
feature_names=shap_sample_X.columns.tolist()
)

shap.summary_plot(explanation_for_class, shap_sample_X.values, plot_type="
    bar", show=False,
feature_names=shap_sample_X.columns.tolist())
plt.title(f"SHAP Feature Importance for '{class_name}' Class")
plt.tight_layout()
plt.savefig(f'shap_summary_plot_class_{i}_{class_name}.png')
plt.show()

shap.plots.beeswarm(explanation_for_class, max_display=15, show=False)
plt.title(f"SHAP Beeswarm Plot for '{class_name}' Class")
plt.tight_layout()
plt.savefig(f'shap_beeswarm_plot_class_{i}_{class_name}.png')
plt.show()

print(f"SHAP plots for class '{class_name}' saved.")

else:  # Binary or effective single output from explainer.shap_values
```

```
print("Plotting SHAP summary plot (could be binary or averaged multi-class)
    ...")
# For overall/binary, the raw shap_values_raw will be a single array [
    n_samples, n_features]

overall_explanation = shap.Explanation(
values=shap_values_raw if isinstance(shap_values_raw, np.ndarray) else
    shap_values_raw[0],
# Take the array
base_values=overall_base_value,
data=shap_sample_X.values,  # Pass as NumPy array
feature_names=shap_sample_X.columns.tolist()
)

shap.summary_plot(overall_explanation, shap_sample_X.values, show=False,
feature_names=shap_sample_X.columns.tolist())
plt.tight_layout()
plt.savefig('shap_summary_plot_overall.png')
plt.show()

shap.plots.beeswarm(overall_explanation, max_display=15, show=False)
plt.title(f"SHAP Beeswarm Plot for Overall Impact")  # Clarified title
plt.tight_layout()
plt.savefig('shap_beeswarm_plot_overall.png')
plt.show()
print("SHAP plots saved as 'shap_summary_plot_overall.png' and '
    shap_beeswarm_plot_overall.png'")

except Exception as e:
# This catch is for initial explainer.shap_values call failing or returning
     something unexpected
print(f"Error computing SHAP values directly: {e}. Attempting fallback with
     explainer(shap_sample_X).")
# Fallback for older SHAP versions or different explainer output
explanation_obj = explainer(shap_sample_X)  # This typically returns an
    Explanation object directly

if isinstance(explanation_obj.values, list) and len(
explanation_obj.values) > 1:  # Multi-class output from fallback
num_classes_to_plot = min(3, len(le.classes_))
print(f"Multi-class SHAP values detected from fallback. Plotting for {
    num_classes_to_plot} key classes...")

expected_values_list = explanation_obj.base_values  # Base values should be
     multi-output if values are
if not isinstance(expected_values_list, (list, np.ndarray)):
expected_values_list = [expected_values_list] * len(le.classes_)
```

```
for i in range(num_classes_to_plot):
class_name = le.inverse_transform([i])[0]
if class_name == 'BENIGN' and len(le.classes_) > 1:
print(f"Skipping SHAP plot for BENIGN class (Index {i}).")
continue

print(f"Plotting SHAP for class: '{class_name}' (Index {i}) from fallback."
   )

# Create a shap.Explanation object for the current class from fallback
current_class_explanation = shap.Explanation(
values=explanation_obj.values[:, :, i],
# Select values for this class from multi-output (N,F,C) -> (N,F)
base_values=expected_values_list[i],
data=shap_sample_X.values,  # Pass as NumPy array
feature_names=shap_sample_X.columns.tolist()
)

shap.summary_plot(current_class_explanation, shap_sample_X.values,
   plot_type="bar", show=False,
feature_names=shap_sample_X.columns.tolist())
plt.title(f"SHAP Feature Importance for '{class_name}' Class (Fallback)")
plt.tight_layout()
plt.savefig(f'shap_summary_plot_class_{i}_{class_name}_fallback.png')
plt.show()

shap.plots.beeswarm(current_class_explanation, max_display=15, show=False)
plt.title(f"SHAP Beeswarm Plot for '{class_name}' Class (Fallback)")
plt.tight_layout()
plt.savefig(f'shap_beeswarm_plot_class_{i}_{class_name}_fallback.png')
plt.show()

print(f"SHAP plots for class '{class_name}' (fallback) saved.")
else:  # Binary or effective single output from fallback explainer
print("Plotting SHAP summary plot (could be binary or averaged multi-class)
    from fallback...")

fallback_shap_values_for_plot = explanation_obj.values
if len(fallback_shap_values_for_plot.shape) == 3:  # If it's (N, F, C),
   average across classes C
print("  Averaging SHAP values across classes for overall fallback plot.")
fallback_shap_values_for_plot = np.mean(np.abs(
   fallback_shap_values_for_plot),
axis=-1)  # Average absolute values for overall impact
# Note: base_values for averaged SHAP might be 0, or average of base_values
   . For simplicity, use first.
fallback_base_value = np.mean(explanation_obj.base_values) if isinstance(
   explanation_obj.base_values,
(list,
np.ndarray)) else explanation_obj.base_values
else:  # Assume (N, F) or (N, )
fallback_base_value = overall_base_value  # Use the pre-determined
   overall_base_value

fallback_overall_explanation = shap.Explanation(
values=fallback_shap_values_for_plot,
base_values=fallback_base_value,
data=shap_sample_X.values,  # Pass as NumPy array
feature_names=shap_sample_X.columns.tolist()
)

shap.summary_plot(fallback_overall_explanation, shap_sample_X.values, show=
   False,
feature_names=shap_sample_X.columns.tolist())
plt.tight_layout()
plt.savefig('shap_summary_plot_overall_fallback.png')
plt.show()

shap.plots.beeswarm(fallback_overall_explanation, max_display=15, show=
   False)
plt.title(f"SHAP Beeswarm Plot for Overall Impact (Fallback)")  # Clarified
    title
plt.tight_layout()
plt.savefig('shap_beeswarm_plot_overall_fallback.png')
plt.show()
print(
"SHAP plots saved as 'shap_summary_plot_overall_fallback.png' and '
   shap_beeswarm_plot_overall_fallback.png'")

print("Evaluation and Interpretability complete!")
return best_xgb_model  # Return model for potential saving


# ---
# MAIN EXECUTION BLOCK

if __name__ == "__main__":
DATA_PATH = 'MachineLearningCVE'  # IMPORTANT: Change this to your actual
   data directory where CSVs are!
RANDOM_STATE = 42
N_FEATURES_TO_SELECT = 60  # Example: Select top 60 features (RQ2) - adjust
    based on EDA/results
DATA_SAMPLING_FRACTION = 0.01  # NEW: Adjusted to 1% for development/
   testing

print("--- Initiating Pipeline ---")
```

```
# Phase 2: Data Acquisition and Pre-processing
# Step 2: Load and Initial Clean Data (RQ1)
# Pass the new sampling_fraction parameter
df_raw = load_and_initial_clean_data(DATA_PATH, sampling_fraction=
    DATA_SAMPLING_FRACTION, random_state=RANDOM_STATE)

# Step 3: Pre-process Data (Imputation, Scaling, Categorical Encoding) (RQ1
    )
X_processed, y_encoded, label_encoder = preprocess_data(df_raw,
    random_state=RANDOM_STATE)

# Step 4: Split Data and Handle Multi-Class Imbalance (RQ1)
X_train_resampled, X_test_original, y_train_resampled, y_test_original = \
split_and_handle_imbalance(X_processed, y_encoded, random_state=
    RANDOM_STATE)

# Phase 3: Model Development & Experimentation
# Step 5: Feature Engineering / Selection (RQ2)
X_train_final, X_test_final = perform_feature_selection(
X_train_resampled, X_test_original, y_train_resampled, y_test_original,
num_features=N_FEATURES_TO_SELECT, random_state=RANDOM_STATE
)

# Step 6: XGBoost Model Training & Hyperparameter Tuning (RQ2)
optimized_xgb_model = train_and_tune_xgboost(X_train_final,
    y_train_resampled, label_encoder,
random_state=RANDOM_STATE)

# Step 7: Comparative Analysis (RQ3)
comparative_results = perform_comparative_analysis(
X_train_final, X_test_final, y_train_resampled, y_test_original,
    label_encoder, random_state=RANDOM_STATE
)

# Phase 4: Result Analysis
# Step 8: Evaluation and Interpretability (RQ1, RQ2, RQ3)
# This step uses the optimized XGBoost model for detailed analysis.
# It also summarizes the comparative results from Step 7.
evaluate_and_interpret(optimized_xgb_model, comparative_results,
    X_test_final, y_test_original, label_encoder,
random_state=RANDOM_STATE)

# ---
# FINAL STEP: SAVE OPTIMIZED XGBOOST MODEL & LABEL ENCODER (for
    reproducibility)
joblib.dump(optimized_xgb_model, 'optimized_xgboost_nids_model.pkl')
```

```
print("\nOptimized XGBoost NIDS model saved to '
    optimized_xgboost_nids_model.pkl'")
# Label encoder is already saved in preprocess_data function.

print("\nXGBoost on NIDS pipeline execution complete!")
```