AGH

AGH University of Science and Technology

FIELD OF SCIENCE ENGINEERING AND TECHNOLOGY

SCIENTIFIC DISCIPLINE INFORMATION AND COMMUNICATION TECHNOLOGY

DOCTORAL DISSERTATION

Elephant Traffic Engineering: Adaptive Routing with Intelligent Flow Classification

Author: Bartosz Kądziołka

Supervisor: Robert Wójcik, PhD, DSc

Completed at: Faculty of Computer Science, Electronics, and Telecommunications

Kraków, 2024

A special thanks to Robert Wójcik, whose guidance and valuable discussions about research directions and results played a crucial role in shaping this dissertation. Also, I would like to express my sincere gratitude to the entire QoS research group, in which I had the pleasure to participate, for their support, insightful feedback, and collaborative spirit throughout this journey. Additionally, I extend my heartfelt thanks to Piotr Jurkiewicz as his insights and feedback on various issues have played a crucial role in enhancing my results. Working with him has been a truly rewarding experience, and I look forward to engaging in many more productive discussions in the future. I would also like to express my gratitude to Michał Wierzbiński for many valuable discussions in the AI/ML field, which have significantly contributed to the development of my work. This work would not have been possible without the collective support of all these individuals.

Abstract

The objective of this dissertation is to demonstrate that integrating Software-Defined Networking (SDN), Machine Learning (ML), and Flow-Aware Multi-Topology Adaptive Routing (FAMTAR) can improve network traffic management and enhance Quality of Service (QoS), with a specific emphasis on early identification and handling of large network flows. To validate this hypothesis, various mechanisms and strategies are proposed, implemented, and evaluated.

The research primarily focuses on two key areas: flow classification and traffic engineering. In flow classification, the study aims to identify elephant flows at the earliest possible stage using only the 5-tuple information with ML models. A comprehensive evaluation of different normalization methods, input data representations, and problem formulations revealed that representing data in terms of bits led to superior accuracy across all tested models. Among these, the best-performing model is the extremely randomized tree classifier, achieving a 66.35 times reduction in the number of flow table operations.

In the area of traffic engineering, a hybrid system integrating ML-based classification with FAMTAR, and SDN was introduced. The proposed system demonstrates improvements in network performance by effectively segregating flow types and optimizing routing strategies. This approach successfully reduces network congestion and enhances resource utilization, leading to a more stable and efficient network. Limiting individual traffic engineering to elephant flows allows for a substantial reductions in Forwarding Flow Table (FFT) size requirements and operations within network switches.

The dissertation's contributions include developing accurate and early flow classification techniques, evaluating and comparing various machine learning models, and successfully integrating machine learning with FAMTAR and SDN. The research demonstrates the clear benefits of applying machine learning in network traffic management, achieving optimized traffic engineering, improved QoS, and a reduction in flow table size and operations. These contributions collectively validate the thesis that integrating SDN, machine learning-based flow classification and adaptive routing can effectively enhance network traffic management and QoS, especially in scenarios with high traffic loads.

Streszczenie

Celem tej rozprawy jest wykazanie, że integracja sieci sterowanych programowo (SDN), uczenia maszynowego (ML) i rutingu adaptacyjnego zorientowanego na przepływy (FAMTAR) może poprawić zarządzanie ruchem oraz zwiększyć jakość usług (QoS) w sieciach komputerowych, ze szczególnym uwzględnieniem wczesnej identyfikacji i obsługi dużych przepływów sieciowych. Aby zweryfikować tę hipotezę, zaproponowano, wdrożono i oceniono różne mechanizmy i strategie.

Badania koncentrują się głównie na dwóch kluczowych obszarach: klasyfikacji przepływów i inżynierii ruchu. W zakresie klasyfikacji przepływów zaproponowano rozwiązanie mające na celu identyfikację największych przepływów (*elephant flows*) na jak najwcześniejszym etapie, wykorzystując jedynie informacje zawarte w nagłówku pakietu (tzw. *5-tuple*) przy użyciu modeli uczenia maszynowego. Kompleksowa ocena różnych metod normalizacji, reprezentacji danych i sformułowań problemów wykazała, że reprezentowanie danych wejściowych w postaci bitów prowadziło do lepszej dokładności we wszystkich testowanych modelach. Spośród tych modeli, najlepiej radzącym sobie był klasyfikator oparty na drzewach ekstremalnie zrandomizowanych, osiągający 66,35 krotną redukcję liczby operacji w tablicach przepływów.

W obszarze inżynierii ruchu zaproponowano hybrydowy system integrujący klasyfikację opartą na ML z FAMTAR i SDN. Zaproponowany system wykazuje poprawę wydajności sieci, skutecznie segregując typy przepływów i optymalizując strategie routingu. Takie podejście skutecznie redukuje przeciążenia w sieci i zwiększa wykorzystanie zasobów, prowadząc do bardziej stabilnej i efektywnej pracy sieci. Ograniczenie indywidualnej inżynierii ruchu jedynie do największych przepływów pozwala na znaczne zmniejszenie wymagań dotyczących rozmiaru tablicy przekazywania przepływywów (FFT) w przełącznikach sieciowych.

Wkład rozprawy w dyscyplinę obejmuje opracowanie dokładnych i wczesnych technik klasyfikacji przepływów, ocenę i porównanie różnych modeli uczenia maszynowego oraz udaną integrację uczenia maszynowego z FAMTAR i SDN. Badania wykazują wyraźne korzyści płynące z zastosowania uczenia maszynowego w zarządzaniu ruchem sieciowym, osiągając optymalizację inżynierii ruchu, poprawę jakości obłsugi oraz redukcję rozmiaru tablic przepływów i operacji na nich. Przedstaiowne wyniki potwierdzają tezę, że integracja SDN, klasyfikacji przepływów opartej na uczeniu maszynowym i adaptacyjnego routingu może skutecznie poprawić zarządzanie ruchem sieciowym i QoS, szczególnie w scenariuszach o wysokich obciążeniach ruchu.

Contents

List of Figures

Chapter 1

Introduction

The total Internet Protocol (IP) traffic has been consistently increasing globally. According to forecasts made before 2023, it was predicted that there would be an additional 1.4 billion Internet users by 2023 compared to 2018 and that the number of networked devices would grow from 18.4 billion in 2018 to 29.3 billion in 2023 [25]. This substantial growth pressures network operators to manage traffic while ensuring guaranteed Quality of Service (QoS), necessitating more efficient and effective utilization of network resources. In addition to the increase in the number of users, the variety of intelligent end devices is also increasing, driven by the proliferation of the Internet of Things (IoT). Ensuring efficient and comprehensive connectivity requires the integration of diverse networking protocols and transmission techniques across both single- and multi-domain scenarios. Modern services such as virtual reality, tactile internet, and vehicular networks contribute to the exponential growth of overall traffic. Not only is the volume of data increasing, but the speed of transmission is also accelerating as core network infrastructure evolves to meet the demands at the edge where technologies such as 5G and Fiber-to-the-Home are providing high-throughput connectivity to end-users.

To achieve the desired Quality of Experience (QoE) for users accessing these novel services, both service providers and network operators must collaborate to optimize various QoS factors in a dynamic and competitive environment. Moreover, efficient management is crucial for balancing operational and capital expenditures, while also ensuring fair service delivery to multiple tenants. Given the complexity of modern network infrastructures, manual management and monitoring by operators and administrators is impractical. Analytical modeling of such systems requires expert knowledge and, in numerous instances, is infeasible due to the vast number of variables and the extensive solution space. Additionally, traditional methods relying on complex mathematical modeling and optimization techniques have become ineffective due to the dynamic nature of the environment, characterized by factors such as increased mobility, the bursty nature of traffic, and multi-tenancy. Although there is a demand for nearly autonomous networks, general approaches face challenges with an overwhelming number of variables and a vast solution space, making them impractical for mathematical methods. Solutions tailored to specific infrastructures often depend on particular topologies, traffic patterns, and technologies, which limits their transferability. As a result, new approaches must be developed to handle these challenges effectively.

In recent years, Software-Defined Networking (SDN) has attracted significant interest in the area of network management. SDN facilitates traffic management within networks by separating the control and data planes. The control plane, typically embodied by a central controller, is responsible for making decisions, particularly regarding packet routing within the data plane. The predominant SDN application today is reactive control, where SDN switches consult the controller for decisions on each new IP flow. However, SDN can also be applied in multi-layer networks, where a single controller manages more than one layer. This approach offers numerous benefits for multi-layer network management. With a comprehensive view of the network environment provided by SDN, the controller can manage the network according to the QoS requirements of incoming traffic, optimizing resource usage. The existence of a single controller greatly simplifies network control and management. Nonetheless, expanding the controller's scope also raises concerns about security and the potential for a single point of failure in the network's control plane.

Traditionally, engineers solved problems with computers by developing detailed analytical or simulation models, which then guided the creation of algorithms — a series of instructions for computer programs. Nowadays, Machine Learning (ML) involves algorithms that generate other algorithms, not by explicit programming, but by learning from historical data. Instead of being directly told how to act, ML algorithms identify and leverage patterns in data to make decisions and predictions. They analyze large datasets to understand system behaviors and use this knowledge to forecast outcomes for similar situations in the future. While ML may not always provide the optimal solution, it often delivers effective results, especially in complex systems where traditional modeling falls short. The main challenges include supplying broad, high-quality training data or a realistic model environment and defining a clear optimization goal with an appropriate metric to measure outcomes. When these elements are well-established, ML can achieve remarkable success across various domains. Emerging technologies and networking concepts provide a favorable environment to address the technical requirements and limitations of ML. For instance, SDN enables centralized knowledge collection about the network in the controller. This allows ML algorithms to be supplied with the input data and propagate future actions across the network. The sophisticated control plane, already well-established in the core network via existing Network Management Systems, is being extended towards access networks through 5G standards and Virtual Network Functions. End nodes possess sufficient intelligence to execute actions prescribed by ML solutions. Additionally, virtualization and cloud-based solutions streamline the training process by offering specialized hardware with Graphical and Tensor Processing Units, distributed data processing frameworks like Hadoop and Spark, and virtually unlimited long-term storage.

Flow-based traffic engineering has recently emerged as an effective method for managing the growing demands on networks while maintaining QoS standards [3] [64] [79]. This approach assigns a specific forwarding entry for each flow in the switch memory, specifying the next hop in the flow's route. This setup allows to establish multiple paths for flows with the same source and destination addresses, enabling multipath routing. Additionally, the routes for incoming flows can be selected based on current or anticipated network congestion, allowing for adaptive routing that avoids congested links. This form of flow-based adaptive routing is recognized for providing greater stability compared to traditional dynamic load balancing in the classic, IP prefix-based, routing [54].

However, a significant challenge arises due to the limited capacity of flow tables in switches, as the number of concurrent flows often surpasses these limits [103]. In SDN, the controller's throughput limitations can also create bottlenecks when handling new flow setups.

To address these issues, one strategy involves dedicating flow table entries exclusively to the largest flows, known as *elephant flows* while routing the majority of smaller, *mouse flows* via default, shortest-path routes. This approach reduces the number of required flow table entries, enhancing packet switching speed and efficiency. Notably, analyses have shown that a small fraction of flows, roughly 0.2-0.4%, accounts for 80% of the total network traffic, a distribution even more skewed than the traditional Pareto principle suggests [53, 78].

The classification of these flows ideally occurs with the initial packet, preventing mid-connection rerouting that could disrupt transport protocol path state estimations. The first packet classification is crucial for efficiently managing controller load and ensuring QoS. This method can improve service within datacenters [117, 72] and across wide-area networks, including inter-datacenter communication [100]. By classifying flows at their inception, network operators can immediately direct traffic to the appropriate queues or paths, allowing for more precise and flow-specific treatment based solely on packet header information. This early classification is particularly critical for applications that require consistent QoS from the start, making it an essential aspect of modern network management.

Identifying the largest flows is a challenging task. The objective is to detect these flows as early as possible so that the network can quickly assign dedicated entries and ensure that most of their packets follow predetermined routing paths. Ideally, this classification should occur with the first packet to avoid the need for rerouting during the connection. This method relies solely on information available in packet headers, such as the 5-tuple (protocol, source, and destination IP addresses, and ports), without using details about packet sequences such as size or interarrival times. Earlier classification also allows a greater share of flow traffic to be subject to flow type-specific treatment. Such classification can be based solely on information contained in packet headers.

1.1 Scope and thesis

This dissertation addresses the challenge of managing increasing network traffic while ensuring the QoS in modern, complex network environments. With global IP traffic continuing to rise, driven by an expanding number of Internet users, the proliferation of intelligent end-devices, and the surge of emerging technologies such as the IoT, 5G, and fiber networks, network operators face significant pressure to optimize their infrastructures. These developments lead to not only an increased volume of data but also demand for faster transmission speeds, more complex traffic patterns, and a diverse range of services requiring consistent and reliable performance. To address these challenges, the research explores how combining SDN, FAMTAR, and ML can improve traffic engineering, specifically focusing on the classification and management of large network flows. The thesis explores how these technologies can be integrated to develop a more efficient and scalable traffic engineering system capable of managing elephant flows, which account for the majority of traffic in networks, but only a small fraction of the total number of flows. By identifying and classifying these large flows at the earliest possible stage – ideally, with the first packet -— network operators can

assign specific routing paths and manage resources more effectively, thus improving the overall network performance and QoS. The proposed approach leverages SDN's centralized control and real-time decision-making, alongside machine learning models to classify flows dynamically based on packet headers. The combination of these methods allows for intelligent traffic management in both core and access networks.

The following thesis is proposed and validated in this dissertation:

It is possible to improve network traffic management and enhance Quality of Service by integrating Software-Defined Networking, Machine Learning, and Flow-Aware Multi-Topology Adaptive Routing, with a specific focus on early elephant flow identification and handling.

All the proposed mechanisms have been implemented, tested, and evaluated in simulated environments, showing their effectiveness in improving traffic management. The results demonstrate that by classifying and treating elephant flows separately, networks can achieve better resource utilization, reduce congestion, and provide a more reliable service to end-users, especially in environments with high data volumes and diverse traffic patterns.

1.2 Publications

Some of the results presented in this dissertation were published in the following papers:

[56] Bartosz Kądziołka, Piotr Jurkiewicz, Robert Wójcik, and Jerzy Domżał. Utilizing tabnet deep learning for elephant flow detection by analyzing information in first packet headers. *Entropy*, 26(7):537, 2024

[66] Bartosz Kądziołka, Piotr Jurkiewicz, Robert Wójcik, and Jerzy Domżał. Elephant flow classification on the first packet with neural networks. *IEEE Access*, 12:65298–65309, 2024

[51] Piotr Jurkiewicz, Bartosz Kadziołka, Mirosław Kantor, Jerzy Domżał, and Robert Wójcik. Machine learning-based elephant flow classification on the first packet. *IEEE Access*, 12:105744–105760, 2024

[52] Piotr Jurkiewicz, Bartosz Kądziołka, Mirosław Kantor, Robert Wójcik, and Jerzy Domżał. Elephant flow detection with random forest models under programmable network dataplane constraints. *IEEE Access*, pages 1–1, 2024

[55] Bartosz Kądziołka and Piotr Jurkiewicz. Redukcja wielkości tablicy przepływów dzięki detekcji dużych przepływów za pomocą uczenia maszynowego. *Przegląd Telekomunikacyjny+ Wiadomości Telekomunikacyjne*, 2023

[14] Piotr Borylo, Edyta Biernacka, Jerzy Domzal, Bartosz Kadziolka, Miroslaw Kantor, Krzysztof Rusek, Maciej Skala, Krzysztof Wajda, Robert Wojcik, and Wojciech Zabek. Neural networks in selected aspects of communications and networking. *IEEE Access*, 2024

[13] Piotr Borylo, Edyta Biernacka, Jerzy Domzal, Bartosz Kadziolka, Miroslaw Kantor, Krzysztof Rusek, Maciej Skala, Krzysztof Wajda, Robert Wojcik, and Wojciech Zabek. A tutorial on reinforcement learning in selected aspects of communications and networking. *Computer Communications*, 208:89–110, 2023

[57] Bartosz Kądziołka, Maciej Skała, Robert WóJCIK, Piotr Jurkiewicz, and Jerzy Domżał. Employing famtar and ahb to achieve an optical resource-efficient multilayer ip-over-eon sdn network. *IEEE Access*, 10:94089–94099, 2022

[56] demonstrates that employing a TabNet model can accurately identify elephant flows right at the start of the flow and makes it possible to reduce the number of flow operations by up to 20 times while still effectively managing 80% of the network traffic through individual flow entries. The same metrics (reduction of flow table operations and traffic coverage) were investigated in [66]. The simple neural network model presented in [66] reduced flow table operations by a factor of 14.7 under the optimal hyperparameter configuration. Much better results for the reduction of flow operations and flow table entries are presented in [51]. Apart from the better results, it is also shown why traditional ML metrics are inadequate for performance assessment in the TE and QoS applications. The work on finding the best model for the TE and QoS metrics continued in [52]. As shown, the best-performing models were tree-based models — the extremely randomized trees. Extremely randomized trees were able to reduce flow operations by a factor of 66.35 and the average number of flow table entries by 19.73 while covering 80% of the traffic. [55] was preliminary research for the described above more in-depth analysis of the various machine learning models aiming to solve the elephant flow classification problem. [55] was presented at the Konferencja Radiokomunikacji i Teleinformatyki (KRiT) 2023 Conference held in Cracow, Poland. Additionally, there were two tutorials on the ML utilized in the computer networks context. Both of them also served as preliminary research for this dissertation - in a very wide context. The paper [14] explores how neural networks are applied to various areas of communications and networking, including traffic prediction, traffic engineering, QoS, QoE, reliability, robustness, and security. In addition to these network aspects, an application-oriented approach is examined, focusing on how machine learning techniques can enhance the deployment of emerging network applications. In [13], a tutorial on Reinforcement Learning (RL) is provided, aiming to deliver foundational knowledge of the RL method and thoroughly study examples of RL-based solutions for addressing challenges in different areas of communications and networking.

Also, a non-ML related research paper was published. [57] is a research focusing on multi-layer SDN architectures were two layer-specific mechanisms (FAMTAR in IP layer, and Automatic Hidden Bypasses (AHB) in optical layer) were managed simultaneously by the SDN controller in order to optimize the QoS.

1.3 Structure of the dissertation

The dissertation is organized into seven chapters, which can be divided into three distinct parts.

The first part (Chapters 1, 2, 3, and 4) focuses on establishing the theoretical background and presenting the state of the art related to the technologies used in the research. Chapter 1 provides an introduction to the dissertation, outlining the research objectives, scope, and an overview of the structure. Chapter 2 introduces the concept of SDN, covering its definition, key benefits, and the challenges it poses in network environments. Chapter 3 explores FAMTAR, explaining its conceptual framework and operational details. This part concludes with Chapter 4, which offers an in-depth overview of machine learning techniques, including supervised, unsupervised, and reinforcement learning. The most widely used models in these approaches are also discussed to provide a comprehensive background for the later stages of the research.

The second part of the dissertation, represented by Chapter 5, is dedicated to the original research conducted on elephant flow classification. This chapter details the experimental approach, covering the various machine learning models used, including neural networks, TabNet, gradient boosting, and trees. Each section

of Chapter 5 examines specific aspects such as input features, datasets, training methodologies, and model performance. The results of this classification research are carefully analyzed and compared, providing key insights into how these techniques can be applied to improve network performance.

The final part (Chapters 6 and 7) builds on the findings from the previous sections and presents a traffic engineering system specifically designed for managing elephant flows. Chapter 6 integrates the concepts of FAMTAR, SDN, and flow classification into a comprehensive solution for elephant-only traffic engineering, describing both the conceptual design and the evaluation of this system. It includes the setup of the experiments and an analysis of the results, demonstrating the effectiveness of the proposed solution. Finally, Chapter 7 concludes the dissertation by summarizing the contributions of the research, and reflecting on its findings.

Chapter 2

Software-Defined Networking

SDN is a very popular technology, as can be seen by the number of results provided by Google Scholar. The keyword "software-defined networking" returns almost 18,000 results since 2023. There are numerous solutions aimed at optimizing resource utilization in multi-layer SDN, but most of them primarily focus on challenges, and security aspects or single-layer optimization (such as implementing bypasses in the optical layer, or better routing in the IP layer) and demonstration on how to manage network control across multi-layer networks. Below are some works that provide various mechanisms and solutions for multi-layer networks.

In [101], the authors explore the integration of Segment Routing (SR), SDN, and optical bypasses. A custom SDN solution manages edge node label stacking and optical bypasses, which are activated in response to load variations. The routing policy for the optical bypass relies on a predefined threshold, eliminating the need for signaling protocols. Segment Routing is further examined in [21], where it is utilized in two scenarios within a multi-layer network: first, in combination with SDN and dynamic optical bypasses, and second, to effectively balance traffic loads, including non-Equal-cost multi-path (ECMP) routes. Additionally, research presented in [34] investigates the use of Segment Routing in a 5G multi-layer, multi-domain network. The study demonstrates the feasibility of SDN-based network slicing for disaggregated 5G transport networks, with slices defined across multiple layers and provisioned across various domains. In [57] FAMTAR, and AHB mechanisms are used simultaneously in a multi-layer SDN network. Those mechanisms are used in parallel with a novel algorithm that is responsible for finding the best possible source-destination pair for bypass and intelligent allocation and release mechanisms for lightpaths in the optical layer.

Some research efforts, such as those in [70, 86, 37], prioritize enhancing network resilience over resource allocation optimization. In [70], the authors tackle the challenge of cross-layer orchestration in response to IP router failures within IP-over-Elastic Optical Network (EON) environments. They introduce a set of multi-layer restoration algorithms designed to minimize operational expenses. The study in [86] explores survivability in IP-over-EON networks, proposing a proactive restoration approach for integrated multi-layer networks. This method demonstrates superior resource efficiency compared to single-layer protection strategies and includes integer linear programming formulations to ensure network survivability in the event of link or node failures. In [37], the authors present a novel SR mechanism to dynamically restore traffic

flows following network failures. Utilizing an SDN controller to map the network topology only when a failure occurs, this approach significantly reduces recovery time.

A hardware-centric method for multi-layer network management is also explored. In [85], the authors utilize P4 switches instead of relying on the SDN controller. The traffic is redirected to optical bypasses once a specified threshold is met. Notably, the study examines two scenarios for utilizing the bypass: either rerouting all packets or only diverting the excess traffic that surpasses the threshold.

Several surveys address various issues related to multi-layer network management. For example, [58] provides a comprehensive overview of solutions that incorporate SDN into multi-layer network architectures. In addition to presenting these solutions, the survey examines their effects on network stability and complexity. Another notable work is [98], which not only highlights the challenges in multi-layer networks but also focuses on network optimization strategies.

2.1 Definition

SDN has emerged as a prominent topic in the Information and Communications Technology (ICT) field over the past decade. Even though it is not a novel concept, it is still popular. In this section, a widely recognized definition of SDN is provided, together with its principal advantages and challenges. Also, an SDN reference model is introduced to serve as the foundation for this paper.

In conventional IP networks, control and data planes are closely integrated within the same network devices, resulting in a highly decentralized architecture. This design was crucial in the early development of the Internet, as it was believed to be the most effective way to ensure network resilience — a key design objective. Indeed, this method has proven quite successful, contributing to significant improvements in network performance, including faster line rates and increased port densities. Conversely, this approach leads to a complex and relatively static network architecture. This complexity and rigidity are why traditional networks are difficult to manage and control. These traits have contributed to a vertically integrated industry where fostering innovation is challenging.

SDN [76] offers the potential to overcome the limitations of existing network infrastructures. By decoupling the network's control logic (the control plane) from the physical routers and switches that handle the traffic (the data plane), SDN disrupts traditional vertical integration. This separation turns network switches into simple forwarding devices. At the same time, the control logic is centralized in a logically centralized controller, making it easier to enforce policies and manage network configurations and updates [61] by being directly programmable. It's crucial to clarify that while SDN advocates for a logically centralized control model, it does not necessarily entail a physically centralized architecture. Indeed, the demands for high performance, scalability, and reliability prevent the adoption of a fully centralized system. Consequently, real-world SDN deployments often feature physically distributed control planes to meet these requirements.

The decoupling of the control plane from the data plane is facilitated through a well-defined Application Programming Interface (API) that connects the switches to the SDN controller. This interface, commonly exemplified by OpenFlow [77], allows the controller to directly manipulate the state of the data plane elements. An OpenFlow switch operates based on a flow table that contains packet-handling rules. Each rule, which can involve actions such as dropping, forwarding, or modifying traffic, matches a specific subset of

the traffic. Through the instructions from the controller application, an OpenFlow switch can assume various network functions, acting as a router, switch, firewall, or other devices like a load balancer or traffic shaper, effectively serving the roles typically associated with a middlebox.

Interestingly, the foundational principles of SDN including the separation of control and data planes, the flow abstraction for forwarding decisions, the logical centralization of network control, and network programmability are not entirely new concepts by themselves [30]. The distinctive feature of the SDN is its ability to enable network programmability by separating the control and data planes. This architecture allows for the control functions to be managed separately from the data flows, ensuring that network management does not interfere with data transmission. Consequently, network intelligence is shifted from the switches to centralized controllers, allowing switches to be controlled externally by software without the need for onboard intelligence. This separation not only simplifies the network environment, making it more programmable but also enhances the flexibility for external software to dictate network operations.

While SDN and OpenFlow initially began as academic projects [77], they have significantly grown in popularity within the industry. Nowadays, most commercial switch vendors have integrated support for the OpenFlow API into their devices. Additionally, the momentum behind SDN was substantial enough to lead major companies such as Google, Facebook, Yahoo, Microsoft, Verizon, and Deutsche Telekom to start the ONF[1] in 2011. The goal of the ONF was to accelerate the development of and promote the SDN technology. However, at the end of 2023, after 12 years of work, the ONF group announced it was transferring its portfolio of projects which include work on broadband networks, 5G mobile networking, and the Programming Protocol-independent Packet Processors (P4) architecture for cloud networking to the Linux Foundation[2].

Over the past few years, a variety of definitions have emerged, each offering its advantages. According to [65] SDN is a network architecture that can be based on four pillars:

- The control and data planes are separated. Control functionality is shifted away from network devices, which are simplified into basic forwarding units.
- Routing decisions are made based on flows rather than destinations. Within the SDN framework, a flow refers to a series of packets traveling from a source to a destination. All packets in a flow are subject to the same service policies at forwarding devices [83][38]. This flow concept standardizes the operation of diverse network devices, such as routers, switches, firewalls, and middleboxes [47], and flow programming grants unparalleled adaptability, only restricted by the capabilities of the flow tables [77].
- The control logic is transferred to an external entity known as the SDN controller or Network Operating System (NOS). The NOS operates on general-purpose server technology and offers the necessary resources and abstractions to manage the programming of forwarding devices from a logically centralized, abstract perspective of the network. Its role is analogous to that of a traditional operating system.

[1] https://opennetworking.org/

[2] https://opennetworking.org/news-and-events/press-releases/onf-merges-market-leading-portfolio-of-open-source-networking-projects-into-the-linux-foundation/

- The network becomes programmable via software applications that run on the NOS and interact with the data plane devices below. This programmability is a core feature.

ONF has proposed a reference model for SDN, depicted in Figure 2.1. This model is structured into three layers: an infrastructure layer, a control layer, and an application layer, each layered sequentially on top of the other. These layers are interconnected through APIs, with the southbound API linking the infrastructure layer to the control layer and the northbound API connecting the control layer to the application layer.

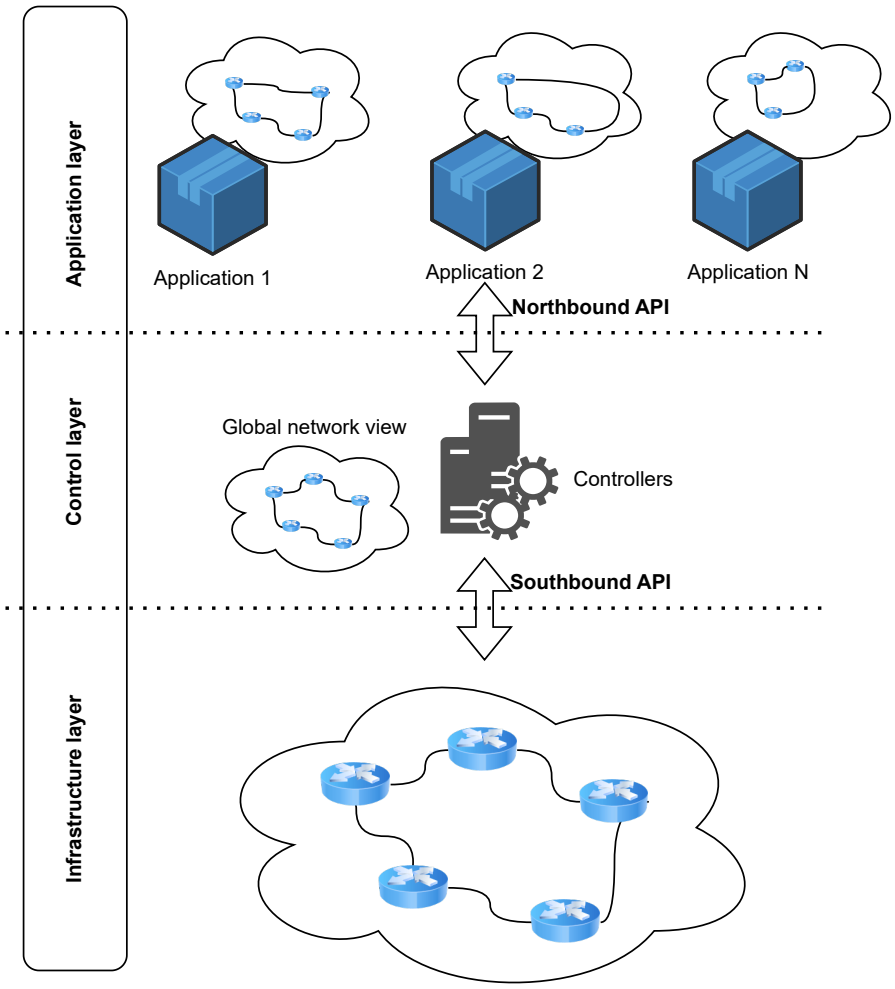

Figure 2.1: SDN architecture based on ONF definition.

The infrastructure layer is primarily made up of forwarding devices such as switches and routers located in the data plane. These devices perform two main functions. Initially, they gather and temporarily store network status information locally before transmitting it to controllers. This status information often encompasses details like network topology, traffic statistics, and network usage. Subsequently, the forwarding devices process packets according to the rules set by a controller.

The control layer is the intermediary between the application and infrastructure layers, facilitated by its two interfaces. The southbound interface allows downward communication with the infrastructure layer,

enabling controllers to utilize the functions of the forwarding devices. These functions typically involve gathering network status and implementing packet forwarding rules. Conversely, the northbound interface facilitates upward communication with the application layer, offering various service access points. Through this API, SDN applications can retrieve network status data from the devices, adjust system settings based on this data, and implement these adjustments by configuring packet forwarding rules on the forwarding devices. Additionally, based on the global network view available in the SDN controller, different network abstractions can be provided to different applications.

The application layer hosts SDN applications that meet specific user needs. Enabled by the programmable platform of the control layer, these applications can manipulate and access the devices found at the infrastructure layer. Examples of such SDN applications include dynamic access control, seamless mobility, and migration, server load balancing, or network virtualization.

2.2 Benefits

The logical centralization of the control logic and decoupling between the control and data planes offers multiple benefits.

- **Improved configuration** - in SDN an entire network can be programmatically configured and dynamically optimized based on network status. This is much simpler and less error-prone than in conventional networks where configuration is mostly manual and dependent on low-level device-specific configurations. Also, significant effort is required to troubleshoot a network with configuration errors - this effort is much lower in the SDN since all configuration is done from a single point, automatically directly via software.

- **Enhanced performance** - SDN enables centralized control, offering a comprehensive view of the network and facilitating feedback between different layers of the network architecture. Consequently, complex performance optimization challenges can be tackled with well-crafted centralized algorithms, functions, and applications. This allows for the development of new solutions to traditional issues such as overall traffic engineering, congestion control, energy-efficient operations, and QoS. These solutions can be swiftly implemented and tested to assess their impact on the network performance.

- **Streamlined Research and Development (R&D)** - SDN facilitates a more straightforward approach to R&D within network environments. By abstracting the control plane from the data plane and centralizing control, researchers can experiment with new algorithms, protocols, and network functionalities without the need for complex configurations on physical devices. This modularity and flexibility enable rapid prototyping and testing of new ideas, which accelerates innovation cycles and reduces the time and cost associated with R&D. Additionally, SDN's programmable nature allows for virtual testing environments that closely mimic real network conditions, providing researchers with a robust platform for iterative testing and refinement.

2.3 Challenges

SDN offers greater flexibility and agility. However, despite the benefits, SDN architecture does not lack a variety of challenges.

- **Flow Table Capacity** - a significant challenge arises due to the limited capacity of flow tables in switches, as the number of concurrent flows often surpasses this limit.
- **Security** - the central controller, which directs traffic and makes decisions for the entire network, becomes a prime target for attacks. If compromised, it can lead to widespread network disruptions or unauthorized data access. Furthermore, the dynamic nature of SDN, with its programmable capabilities and automated systems, can sometimes obscure the visibility of security threats, making it difficult to detect and respond to attacks promptly.
- **Single Point of Failure** - the centralized nature of the SDN controller also introduces the risk of a single point of failure. If the controller experiences a hardware failure, software crash, or targeted attack, the entire network could potentially go down. This reliance on one central node for network decisions and traffic management requires robust failover mechanisms and redundancy strategies to ensure network reliability and uptime in the event of a controller failure. This is critical for maintaining continuous network operation and preventing service disruptions.
- **Legacy Systems** - integrating SDN within existing network infrastructures that include legacy hardware and non-SDN protocols presents significant challenges. Ensuring interoperability between traditional and SDN-enabled components often requires additional transitional tools or layers, which can complicate the network architecture and potentially introduce new points of failure or security vulnerabilities.
- **Network abstraction** - while SDN aims to simplify network management, the abstraction it introduces can also obscure underlying network behaviors, making troubleshooting more complex. Network operators may find it challenging to diagnose issues when they cannot directly access or visualize the physical path of network traffic as easily.

Chapter 3

Flow-Aware Multi-Topology Adaptive Routing

3.1 Concept

FAMTAR [110] was created to address the inefficiencies in traditional routing protocols, such as Open Shortest Path First (OSPF), which tend to use a single optimal path for routing traffic between two endpoints, even when that path becomes congested. As the demand for network resources, particularly bandwidth, continues to increase with the evolution of the Internet, these traditional protocols are unable to adapt dynamically to changes in network conditions, leading to suboptimal utilization of available resources.

FAMTAR was designed to overcome these limitations by operating above the intra-domain routing protocol and cooperating with existing protocols. It introduces a more dynamic approach where, in case of congestion on the optimal path, new traffic flows are redirected to alternative paths, while ongoing flows continue on their current routes. This adaptability ensures more efficient use of network resources by dynamically creating and removing alternative routes as needed.

The development of FAMTAR leverages the increased computational power available in modern network devices, making it possible to manage flow-based traffic more effectively and utilize multiple paths between endpoints. By doing so, FAMTAR can significantly increase the amount of traffic a network can handle, reduce delays, and improve overall network performance. Thus, FAMTAR was created to enhance the efficiency and adaptability of routing protocols in response to the growing demands on network resources.

The main concept behind FAMTAR is the management of network traffic based on the concept of *flows* [109]. In networking, a flow refers to a stream of packets that belong to a specific transmission between two end users. These packets are identified by certain header fields, often referred to as the *5-tuple*, which includes the source and destination IP addresses, source, and destination port numbers, and the transport layer protocol (such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP)) used for transmission. By focusing on flows, FAMTAR can manage and route traffic more efficiently. Instead of treating each packet independently, FAMTAR recognizes and handles flows as continuous streams. This allows it to adaptively reroute new flows to alternative paths in the network when the primary path becomes congested

while keeping ongoing flows on their original paths. This flow-based approach is key to FAMTAR's ability to optimize network resource usage and improve overall traffic management.

3.2 FAMTAR operation

FAMTAR operates by incorporating an additional component into standard router functionality known as the Flow Forwarding Table (FFT). The FFT is central to how FAMTAR processes packets and manages network flows.

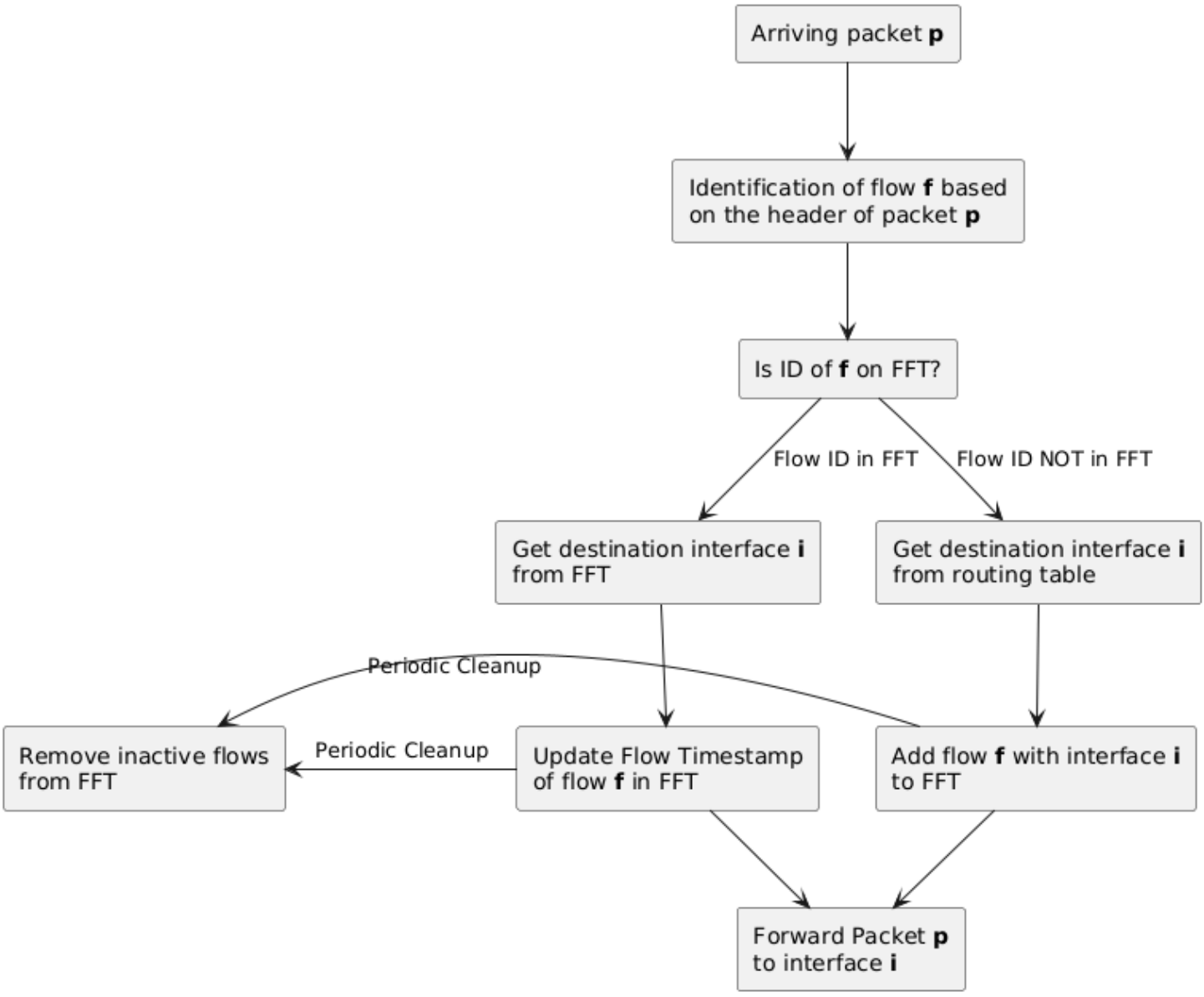

Figure 3.1: Operation of FAMTAR.

The processing of a packet in a FAMTAR router is presented in Figure 3.1. The block diagram from Figure 3.1 is also explained in depth below.

1. **Flow Identification** - when a packet from a new flow arrives at a FAMTAR-enabled router, the router computes a unique Identifier (ID) for the flow. This ID is a hash generated from the 5-tuple of the packet.

2. **Flow Lookup** - the router then checks if this flow ID is already present in the FFT.

    - If the flow is found in the FFT, the router retrieves the corresponding outgoing interface from the FFT, updates the timestamp associated with the flow to reflect the current time, and forwards the packet accordingly. Importantly, in this case, the routing table is not consulted because the FFT already contains the necessary routing information.
    - If the flow is not found in the FFT (indicating it is a new flow), the router determines the appropriate outgoing interface by consulting the routing table, as would normally happen in standard routing procedures. Once the interface is determined, the flow is added to the FFT along with the interface information for future reference.

3. **Flow Maintenance** - the FFT also keeps track of the time when the last packet of each flow was processed. This information is crucial for managing the size of the FFT by allowing the router to remove inactive flows, thereby preventing the table from becoming too large and ensuring scalability.

In FAMTAR, the network operates differently depending on whether there is congestion or not on the network links. **When there is no congestion** the network operates in a classic, traditional way using the existing routing protocol to determine the optimal paths for data transmission. All traffic flows follow the optimal path determined by the routing protocol between the source and destination. The routing protocol ensures that traffic takes the shortest or least-cost path available, as defined by the network topology and link costs. **When there is a congestion** the adjacent router updates the cost of this link with a predefined high value. The detection of congestion can be based on various indicators such as link load, queue occupancy, or packet queuing delay. The router communicates the increased cost of the congested link to other routers in the network as part of a standard topology change message. This is processed by the routing protocol, which then recalculates the optimal paths for data transmission across the network. Because the cost of the congested link is now very high, it is no longer considered part of the optimal path for new traffic flows. The routing protocol will avoid using this link when computing new routes, preferring other available paths unless no alternatives exist. As a result, new traffic flows will be routed through less congested links. Flows that were already active and using the congested link before the congestion was detected will continue to use that link. This ensures that ongoing transmissions are not disrupted or rerouted mid-stream. Once the congestion subsides and the link returns to normal load levels, the router restores the original cost of the link. This allows the link to be considered as part of the optimal paths once again. Future traffic can then resume using this link if it becomes part of the most efficient route.

In summary FAMTAR can efficiently handle network traffic by leveraging flow-based routing, reducing the need to consult the routing table for each packet, and dynamically adapting to network conditions. By storing and reusing flow information, FAMTAR enhances the speed and efficiency of packet forwarding. In FAMTAR FFT is used to execute the majority of the packet routing tasks. Entries in the FFT are static and do not reflect routing table changes. FAMTAR's approach ensures efficient utilization of network resources

by dynamically adjusting routes in response to congestion, thus minimizing traffic delays and preventing further congestion.

Chapter 4

Machine Learning

ML, first defined by Arthur Samuel in 1959 as "the field of study that gives computers the ability to learn without being explicitly programmed," has evolved significantly, yet this foundational definition remains relevant. The general workflow of ML techniques follows a structured process. It begins with the collection or generation of input data, known as training data, which serves as the foundation for the learning process. During the feature extraction stage, the raw data undergoes preprocessing where noise is filtered out, dimensionality is reduced, and critical features are highlighted. This step is crucial as it aims to reduce computational complexity and enhance the accuracy of the model. Following this, the algorithm engages in the learning phase, where it identifies and understands the complex dependencies that characterize the problem at hand. It is important to note that the specific interpretation of data collection, feature engineering, and model learning varies depending on the learning paradigm employed by the ML method. These steps collectively form the process of model creation in ML.

This model can then operate on new data to infer an outcome. The interpretation of this inference and any necessary preprocessing of new data depends on the specific problem being addressed. The inferred outcome could simply provide information to a user, or it might initiate further actions. Ultimately, based on the validation of the inferred outcome, it may be possible, though not required, to refine and enhance the initial phases: data collection, feature engineering, and model learning. This workflow highlights that ML models are particularly well-suited for problems where a substantial amount of representative data is available [16].

The fundamental classification of ML algorithms is based on the learning paradigm employed by each method. The characteristics of these approaches are briefly outlined. There are three main approaches as shown in Figure 4.1. Those approaches are also summarized in Table 4.1.

As mentioned in Chapter 2, SDN is a very popular technology. Saying the same about the Artificial Inteligence (AI) would be an understatement. The recent surge in AI is fueled by significant breakthroughs in machine learning, data analytics, and computing power, which have expanded AI's capabilities and applications. AI has become increasingly popular due to its transformative effects across numerous industries, including healthcare, finance, entertainment, and transportation. Its ability to analyze vast amounts of data, derive insights, and automate complex processes has made it a valuable asset for driving innovation and enhancing efficiency. Moreover, the growing availability of data and advancements in algorithms have led

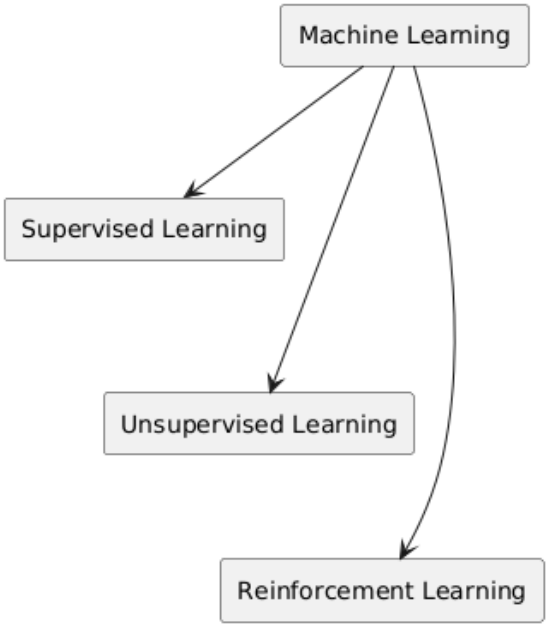

Figure 4.1: ML classes.

to the creation of more sophisticated AI systems, encouraging their widespread adoption and integration into everyday technology and business practices. AI trend is also clearly visible in the communications and networking area [82, 22].

In the domain of AI, many tutorial and survey papers provide in-depth analysis of various AI mechanisms applied in communications and networking. The paper [116] serves as a tutorial on the application of Neural Network (NN) and Deep Learning (DL) in wireless networks. Similarly, [23] offers a comprehensive overview of various NN solutions specifically designed for wireless applications. Additional examples of related work in wireless networks include [1] and [45]. Surveys [107] and [8] are also focused solely on wireless networks, however, they both analyze the RL. The survey by [107] delves into the use of model-free learning for resource management and scheduling within cognitive wireless networks. Conversely, the research reviewed in [8] focuses exclusively on applying RL for the efficient and reliable transmission of data in ad-hoc wireless networks. The survey [112], also presented in a tutorial style, concentrates on the application of NN techniques within SDN. This work covers topics similar such as routing, QoS, and traffic prediction. Paper [14] presents how NN are applied to various aspects of communications and networking, including traffic prediction, traffic engineering, QoS, QoE, reliability, robustness, and, security. Besides network aspects, authors also analyze the application-oriented approach, i.e., how ML techniques are used to boost the deployment of emerging network applications. In [13], a tutorial on RL is presented. Its purpose is to provide fundamentals regarding the RL method and to comprehensively study the examples of applying RL-based solutions to solve problems in different aspects of communications and networking. There are more RL-oriented surveys such as [92], [35], and [105] which explore different facets of communications and networking. Specifically, [92] examines how RL can address challenges in a network caching, task offloading, routing, scheduling, and resource allocation within the frameworks of mobile edge computing, SDN, and network virtualization in 5G environments. [35] reviews studies that apply RL techniques to

achieve efficient autoscaling within cloud environments, whereas [105] surveys solutions using RL to defend against IoT attacks. A very recent, and interesting study [9] offers a review of existing network classification techniques, including methods like port-based identification, deep packet inspection, statistical features combined with ML, and DL algorithms.

The study in [6] presents a fully distributed *multi-layer* routing approach using a BIO-inspired ant colony optimization algorithm for real-time management of both optical and IP/Multiprotocol Label Switching (MPLS) layers. Unlike centralized SDN systems, which depend on a single control point, this method operates with distinct control planes for optical and IP layers, where each network node only has access to local routing data. In contrast, [74] demonstrates a reinforcement learning algorithm applied within the SDN controller, optimizing virtual multi-layer network resource allocation while ensuring rigorous service isolation. This control loop significantly improves resource utilization across network nodes.

Precise traffic prediction is essential for energy conservation and effective network resource management. Utilizing traffic forecasting in core network equipment can lead to substantial power savings. Moreover, accurate forecasting enhances QoS by enabling more efficient resource allocation based on predicted network traffic. In the paper [10] authors aim to solve the traffic matrix prediction problem by adapting Long short-term memory (LSTM) Recurrent Neural Network (RNN). Traffic forecasting in cloud data centers using EON technology is discussed in [2]. Authors present two approaches for traffic prediction that employ ML techniques such as Monte Carlo Tree Search (MCTS) and NN. In [91], the authors tackle the challenge of predicting flow sizes and identifying exceptionally large flows (elephant flows). The study employs NN, Gaussian Process Regression (GPR), and online Bayesian Moment Matching (oBMM) for these predictions. Additionally, [91] demonstrates how routing can be enhanced when these predictors are integrated with SDN.

Traffic engineering refers to the process of optimizing the performance of operational networks. It involves managing and controlling data flows within and between networks to efficiently utilize available network resources, improve network reliability, and ensure a high QoS. Techniques include routing protocol adjustments, bandwidth allocation, and prioritizing traffic based on type and importance to avoid congestion and maximize throughput. The study in [69] introduces a deep learning approach to enhance routing protocols. It presents the QoS-aware Adaptive Routing (QAR) algorithm, an advancement of RL that incorporates a reward system tailored to QoS and individual user needs. Additionally, the article examines a substantial SDN network topology aimed at enhancing data transportation and reducing signaling delays.

Congestion control in networking refers to mechanisms that prevent network traffic from exceeding the capacity of the network, ensuring efficient data flow and preventing packet loss or delay. Algorithms that serve as alternatives or enhancements to the TCP congestion control framework are presented in two significant studies, [48] and [68]. They demonstrate advancements in TCP performance under diverse network conditions. The former employs basic RL strategies, while the latter utilizes advanced Q-learning methods to enhance efficiency and scalability. Subsequently, studies focused on optimizing network admission control to use network infrastructure efficiently and cost-effectively are discussed in the research in [11] and [120]. [11] applies RL approaches to manage resource allocation challenges for Infrastructure Network Provider

(InP), aiming to maximize profits. In contrast, [120] specifically targets spectral resources, developing solutions for dynamic multi-channel access to assign channels to users optimally.

TE mechanisms improved by NN-based solutions can optimize transmission in a network. Prediction models as well as TE algorithms can be a part of the advanced *QoS and QoE* solutions. QoS gauges a network's capability to fulfill service expectations by evaluating performance metrics such as bandwidth, latency, error rate, and uptime. In contrast, QoE assesses the overall satisfaction or dissatisfaction of users with an application or service. While QoE encompasses the complete service experience, akin to user experience in other domains, it is specifically rooted in telecommunications. The QoE analysis in SDN is presented in [73]. The authors introduce a classifier capable of detecting anomalies in real time and forecasting them for the immediate future. This QoE classifier is a component of a QoE prediction system, designed to function as a quality monitoring service at the edge processing layer. For that purpose, three different ML models were tested. Another QoE related research is presented in [104]. The goal of this work is to predict QoE metrics in an environment based on Deep Neural Network (DNN). The study in [94] focuses on establishing a stable control environment within a smart grid, which necessitates fulfilling service demands without experiencing packet loss or time delays. The paper formulates an optimization problem grounded in scheduling challenges and QoS requirements. This problem is then transformed into a Markov Decision Process (MDP) framework to facilitate the development of the RL process.

4.1 Supervised learning

Supervised learning is a fundamental category within ML that involves training algorithms using labeled datasets. In this approach, each data sample in the training set is paired with a corresponding label that provides the correct interpretation or desired outcome for that sample. These labels act as a form of supervision, guiding the learning process as the algorithm maps inputs to outputs. The objective of supervised learning is to enable the model to make accurate predictions when presented with new, unseen data. One of the most common applications of supervised learning is in classification tasks, where the algorithm learns to categorize new data into one of several predefined classes. For instance, a supervised learning model might be trained to classify flows as either *mice* or *elephant*, based on the labeled examples it has seen during training. This ability to classify discrete outcomes is widely used in various fields, from image recognition to spam detection and many, many others. However, supervised learning is not limited to classification tasks. It is also employed in regression tasks, where the goal is to predict a continuous output. For example, a regression model might be used to forecast a flow size in bytes based on flow features such as the 5-tuple of the flow's first packet. In this case, the model learns to output a continuous value rather than a discrete category. NNs, a powerful and flexible type of a supervised learning algorithm, are particularly effective at handling complex tasks that involve large and intricate datasets. These networks, through layers of interconnected neurons, learn to recognize patterns and relationships within the data, making them suitable for tasks ranging from image and speech recognition to natural language processing.

Overall, supervised learning's reliance on labeled data makes it a powerful tool for predictive modeling in scenarios where clear examples of correct outcomes are available. Its ability to handle both classification and regression tasks has made it a cornerstone of modern ML applications.

4.2 Unsupervised learning

Unsupervised learning represents a second major category of ML methods, distinguished by its ability to operate without the need for labeled data. Unlike supervised learning, where the algorithm is guided by predefined labels, unsupervised learning algorithms explore and analyze the internal structure of the data independently. This capability is particularly advantageous in situations where obtaining labeled data is difficult, time-consuming, or expensive. One of the primary tasks that unsupervised learning excels at is clustering. Clustering involves grouping similar data points based on their inherent characteristics. For example, in a video streaming service, unsupervised learning algorithms can analyze user behavior to identify distinct groups of clients with similar viewing habits. These clusters can then be used for personalized recommendations, targeted marketing, or understanding customer segments without any prior knowledge of the groupings. Another key application of unsupervised learning is association, which involves discovering patterns and relationships within the data. Association techniques are designed to identify rules that describe how certain data points are related to each other. For instance, an unsupervised learning model might analyze network traffic patterns and identify clusters of devices that exhibit similar communication behaviors. These clusters could represent different types of users, such as employees accessing internal resources versus guests browsing the internet. Additionally, the model could uncover associations, such as the tendency for certain devices to frequently communicate during specific times of day, which could be valuable for optimizing network resources, enhancing security protocols, and predicting future network usage patterns based on past interactions. NNs, which are also utilized in unsupervised learning, bring additional power and flexibility to this approach. They can learn complex patterns and structures in data without any explicit supervision, making them well-suited for tasks like anomaly detection, data compression, and feature learning.

The absence of the need for labeled data makes unsupervised learning particularly useful for exploring and understanding large, unlabeled datasets. It is a crucial tool in data mining, allowing organizations to uncover hidden patterns and insights that would otherwise remain obscured. By enabling the discovery of natural groupings and associations within data, unsupervised learning helps unlock valuable information, driving informed decision-making across various domains.

4.3 Reinforcement learning

Reinforcement learning represents a unique and dynamic category of ML, distinguished by its focus on learning through interaction with an environment. In this paradigm, the ML algorithm is embodied by an entity known as an agent. Unlike supervised learning, which relies on labeled data, or unsupervised learning, which uncovers patterns without guidance, RL operates without direct supervision. Instead, the agent learns by actively engaging with its environment, making decisions, and observing the consequences of those decisions. The core of RL lies in the concept of rewards and penalties. As the agent interacts with its environment, it takes actions that lead to specific outcomes. For each action, the agent receives feedback in the form of rewards or penalties, which serve as signals guiding its learning process. The agent's ultimate goal is to maximize its total reward by identifying the optimal sequence of actions. This process involves

balancing immediate rewards with long-term benefits, making RL particularly suited for complex decision-making tasks. One of the key strengths of RL is its ability to learn from a vast number of parallel sequences of decisions. Thanks to powerful computing infrastructure, RL algorithms can process and analyze multiple scenarios simultaneously, allowing the agent to refine its decision-making strategy over time. This capability is especially valuable in environments where real-time decisions are critical, such as in game-like scenarios or situations that require continuous adaptation to changing conditions. A prime example of RL in computer networks is its application in dynamic routing optimization. In such a scenario, an agent could be responsible for managing data packet routing within a network. The agent makes decisions on which paths packets should take, based on real-time network conditions such as congestion, latency, and link failures. Each decision directly impacts the overall network performance, and the agent receives rewards for actions that improve efficiency, such as reducing latency or avoiding congested paths. Over time, the agent learns the best strategies to optimize network traffic flow, adapt to changing conditions, and ensure smooth data transmission across the network.

Overall, RL is best suited for problems where an agent must make a series of decisions in an uncertain environment, and where the goal is to achieve the highest possible reward through strategic actions. Its capacity to learn from experience and adapt to new situations makes it an essential tool for applications requiring intelligent and autonomous decision-making.

Table 4.1: Summary of ML classes.

Type of ML	**Description**	**Common Applications**
Supervised Learning	Trained on labeled datasets to map inputs to outputs.	Classification, Regression.
Unsupervised Learning	Analysis of unlabeled data to discover hidden patterns and relationships.	Clustering, Association.
Reinforcement Learning	Involves an agent interacting with an environment to maximize cumulative rewards through sequential decision-making.	Dynamic Routing, Autonomous Systems, Game AI.

4.4 Most popular models

Among the various ML models, NNs, and Tree-Based Algorithms are among the most popular. NNs, particularly deep learning models like Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), have gained widespread popularity due to their exceptional ability to model complex, non-linear relationships in data, making them indispensable for tasks such as image recognition, natural language processing, and autonomous systems.

Tree-based algorithms like Random Forests and Gradient Boosting Machine (GBM) are also highly popular, largely because of their robustness, interpretability, and strong performance across a wide range of classification and regression tasks. These algorithms excel in scenarios with structured data and are favored for their ability to handle both large datasets and high-dimensional data effectively.

Neural Networks in the context of ML were first introduced in the 1940s. The foundational concept dates back to 1943 when Warren McCulloch, a neurophysiologist, and Walter Pitts, a logician, proposed a model of artificial neurons in their seminal paper [75]. This model laid the groundwork for the development of Artificial Neural Network (ANN) by demonstrating how networks of simple units (neurons) could perform logical operations. The next major milestone came in 1958 when Frank Rosenblatt developed the Perceptron [97], the first algorithmically implemented NN capable of learning. The perceptron was a simple single-layer NN and marked the beginning of practical NN research. The revival of NNs occurred in the 1980s with the development of the backpropagation algorithm by researchers including David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams [99]. Backpropagation made it feasible to train multi-layer NNs, which could learn complex, non-linear relationships in data. This advancement sparked renewed interest and led to significant progress in the field, laying the foundation for the deep-learning models that are widely used today.

An ANN [44] (often referred to as a NN) is composed of artificial neurons, which are inspired by the biological neurons found in the human nervous system, and the connections between them. These connections transmit the output of one neuron as the input to another, with each connection assigned a weight that indicates its significance. It is important to note that a neuron can have multiple input and output connections. Neurons within an ANN are organized into layers, with the input layer being the first, responsible for receiving and initially processing the data. The final layer, known as the output layer, generates the network's results. Between the input and output layers, there can be zero or more hidden layers that handle more complex data processing tasks.

To determine the output of a neuron in a hidden layer, the network combines all its inputs and corresponding weights into a weighted sum, as shown in equation (4.1). During the training phase, the network updates these weight values based on the chosen method, with *backpropagation* being one of the most commonly used techniques. In backpropagation, the error from the output is propagated backward through the network, adjusting the weights to minimize the error. The weighted sum is often referred to as the neuron's activation. An activation function produces the neuron's output. It's important to note that activation functions are typically non-linear, such as the sigmoid or hyperbolic tangent functions, and they are only applied in the hidden and output layers.

$$z = \sum_{i=1}^{n} w_i \cdot x_i + b \tag{4.1}$$

where:

z - the weighted sum of the neuron,

n - the number of neuron inputs,

w_i - the weight of the i-th input,

x_i - the value of the i-th input,

b - the bias of the neuron.

Feedforward Neural Network (FFNN) is the earliest and simplest architecture of the ANN. In an FFNN, the connections between the neurons do not form any cycles, meaning there are no feedback loops present in the architecture. Essentially, information flows in a single direction—from the input layer, through any optional hidden layers, and finally to the output layer. The architecture of a basic FFNN is illustrated in Figure 4.2. Typically, when references are made to NN or ANN, they are often referring to FFNNs.

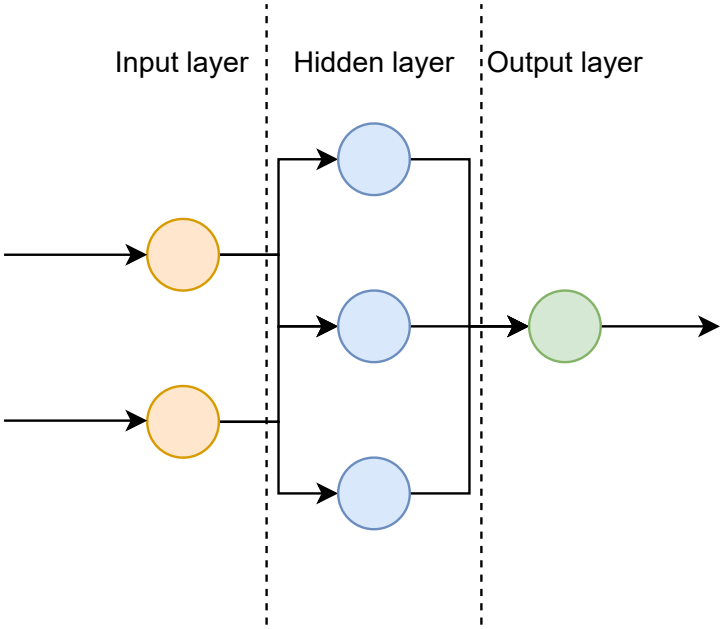

Figure 4.2: FFNN architecture based on common designs found in literature.

DNN, refer to ANN that have multiple hidden layers between the input and output layers - as shown in Figure 4.3. The term "deep" indicates that the network has a greater number of these hidden layers, making it capable of modeling more complex patterns in data. This applies to both FFNNs and RNNs, where having multiple hidden layers makes them Deep FFNNs and Deep RNNs, respectively. The depth of a network — the number of hidden layers — enables it to capture more intricate relationships within the data, which can lead to higher accuracy in predictions. However, there isn't a strict definition of how many layers are needed for a network to be considered "deep", and the optimal number of layers often depends on the specific problem being solved. As the number of layers and neurons in a network increases, the network becomes more powerful in terms of accuracy but also more prone to *overfitting*. Overfitting occurs when a network

becomes too complex and starts learning the noise in the training data, leading to poor performance on new, unseen data. Conversely, a network with too few layers may suffer from *underfitting*, where it cannot capture the underlying patterns in the data, resulting in poor performance across all datasets. Choosing the right number of hidden layers is often based on intuition or through experimental methods where different network architectures are tested. Larger deep networks may also require a distributed approach [39] to handle the computational load, as they can be resource-intensive. Understanding the balance between network depth, accuracy, and the risk of overfitting or underfitting is crucial in designing effective deep networks.

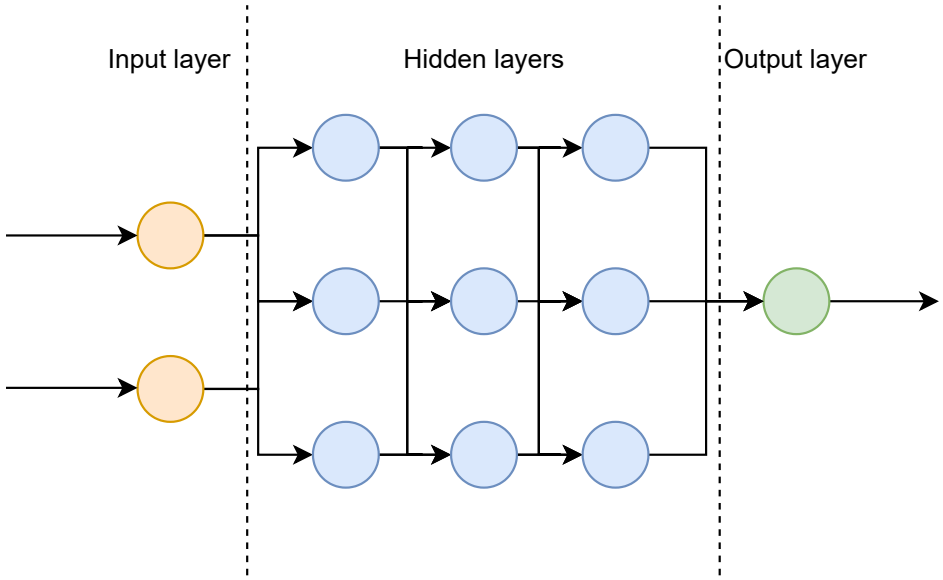

Figure 4.3: DNN architecture based on common designs found in literature.

The *RNN* [96] features a recurrent connection within its hidden layer neurons. The output from each neuron is not only passed on to the next layer but also fed back into the neuron itself (recurrent layer) to capture the sequential information in the input data. The mathematical formulation of the neuron state varies across different RNN architectures, such as LSTM and Gated Recurrent Unit (GRU). Equation (4.2) illustrates the neuron state at time t for the most basic RNN architecture. When making decisions, an RNN considers both the current input and the information it has learned from previous inputs. This type of neural network is particularly useful when context is essential for predicting an outcome, meaning that decisions from earlier iterations or samples can impact the current prediction. RNN architecture is presented in Figure 4.4.

$$h_t = \sigma(W_h \cdot x_t + U_h \cdot h_{t-1} + b_h) \tag{4.2}$$

where:

h_t - the hidden state at time step t,

x_t - the input at time step t,

h_{t-1} - the hidden state from the previous time step $(t-1)$,

W_h - the weight matrix for the input at time step t,

U_h - the weight matrix for the hidden state from the previous time step,

b_h - the bias of the neuron,

σ - the activation function (e.g., sigmoid, tanh).

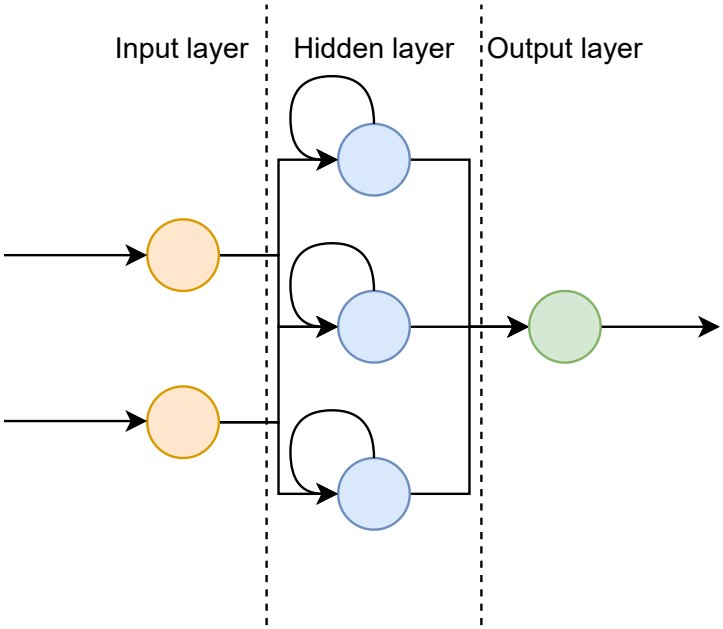

Figure 4.4: RNN architecture based on common designs found in literature.

CNN is a specialized class of DNN that is particularly effective for processing structured grid-like data, such as images. The name "Convolutional" comes from the mathematical operation called convolution, which is a key component of the network's function. As shown in Figure 4.5 a CNN can be divided into two main parts:

1. Feature extraction part:

   - **Convolutional layer** - this is the core layer of a CNN, responsible for extracting features from the input data. The convolutional layer performs a convolution operation, where the input data is combined with a small matrix called a kernel or filter - shown in equation 4.3. This filter slides over the input data, applying the convolution operation to produce an output known as a feature map. The kernel is typically smaller in height and width compared to the input data, allowing it to detect specific features like edges, textures, or patterns.
   - **Pooling layer** - following the convolutional layer, a pooling layer is often applied to reduce the dimensionality of the feature maps, thereby simplifying the computational complexity for subsequent layers. Pooling helps in down-sampling the data while preserving its important characteristics. The most common type of pooling is Max Pooling (shown in equation 4.4), which divides the input matrix into smaller sections and outputs the maximum value from each section.

Other pooling methods include Average Pooling, which calculates the average value, and Sum Pooling, which sums the values in each section.

2. Classification part:

   - The final part of a CNN is typically a fully connected layer, similar to those used in standard ANN. This layer takes the features extracted by the convolutional and pooling layers and uses them to classify the input data into one of several possible classes. The fully connected layer essentially makes the final prediction based on the features identified in the earlier stages.

$$Y_i = (X * K)_i = \sum_{j=1}^{m} \sum_{k=1}^{n} X_{i+j-k} \cdot K_{j,k} \tag{4.3}$$

where:

Y_i - i-th output element of the convolutional layer,
X - input data to the convolutional layer,
K - convolutional kernel (filter),
m, n - the dimensions of the convolutional kernel.

$$Y_{i,j} = \max(X_{2i,2j}, X_{2i,2j+1}, X_{2i+1,2j}, X_{2i+1,2j+1}) \tag{4.4}$$

where:

$Y_{i,j}$ - i, j-th output element of the pooling layer,
X - input data to the pooling layer,
$X_{2i,2j}$ - input element at position $(2i, 2j)$,
$X_{2i,2j+1}$ - input element at position $(2i, 2j + 1)$,
$X_{2i+1,2j}$ - input element at position $(2i + 1, 2j)$,
$X_{2i+1,2j+1}$ - input element at position $(2i + 1, 2j + 1)$.

NNs offer powerful tools for solving a variety of network-related problems. FFNNs are effective for tasks like detecting anomalies, and identifying services and protocols. RNNs are particularly useful for recognizing sequential patterns in network traffic and handling time-dependent issues, such as predicting future network loads or latency. CNNs excel at pattern recognition, making them valuable for analyzing network graphics and detecting visual patterns. These NN architectures can be combined or configured in different ways to address more complex network-related challenges, with the choice of method depending on the nature of the input data, the problem's characteristics, and the desired output.

Tree-based models have gained widespread popularity in the field of ML due to their intuitive structure, versatility, and robust performance across a wide range of tasks. These models are particularly valued

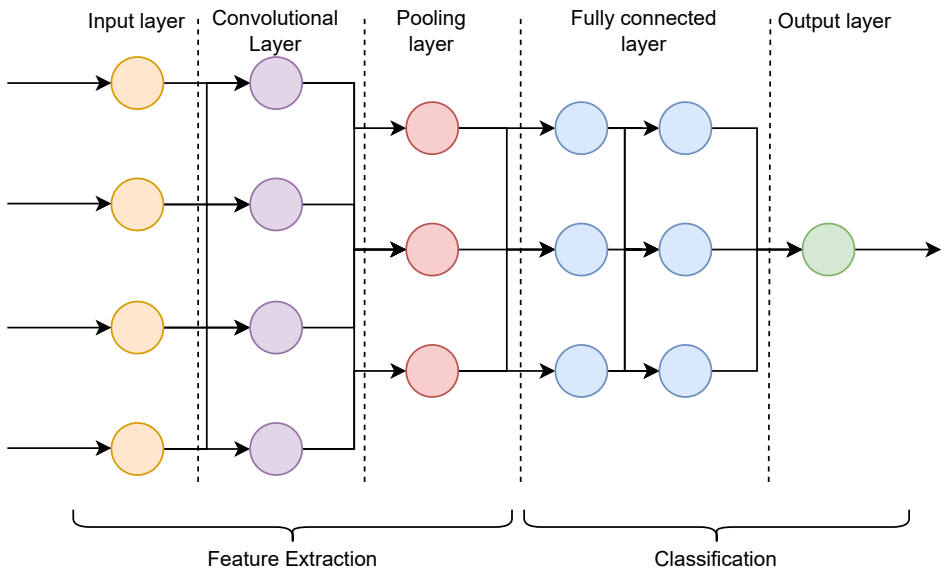

Figure 4.5: CNN architecture based on common designs found in literature.

for their ability to handle both classification and regression problems effectively. One of the key reasons for their popularity is their interpretability — tree-based models mimic human decision-making processes, making it easy for users to understand how predictions are made. Additionally, they require minimal data preprocessing and can handle both numerical and categorical data, making them highly adaptable to various types of datasets. Another significant advantage of tree-based models is their inherent capability to model complex, non-linear relationships within data. This makes them especially useful in real-world applications where relationships between variables are not strictly linear. Furthermore, methods like random forests and gradient boosting, which build multiple trees and aggregate their predictions, offer enhanced accuracy and robustness by reducing the risk of overfitting and improving generalization to new data. The combination of interpretability, flexibility, and strong predictive power has made tree-based models a go-to choice for many practitioners, particularly in many areas of interest, where understanding the model's decision-making process is as important as achieving high accuracy. These qualities ensure that tree-based models remain a cornerstone of ML, providing reliable and insightful solutions across a diverse array of applications.

Decision Tree foundation algorithm - Automatic Interaction Detection (AID) - was first introduced in 1963 by John W. Morgan and John N. Sonquist [81]. This was one of the first algorithms designed to automate the process of decision tree construction. AID was developed primarily for social science research and was used to identify interaction effects in data by partitioning the data into subsets based on the values of independent variables. The AID algorithm was foundational because it introduced the concept of splitting data into homogenous groups based on decision rules, which is a core principle of modern decision tree methods. Although AID was initially focused on identifying interactions in statistical models, it set the stage for later developments in decision tree algorithms [95] that are widely used in ML today.

Decision Trees are a fundamental and intuitive form of tree-based models that play a crucial role in the field of ML and data analysis. At their core, decision trees operate by recursively splitting data into subsets based on the most significant features, which are the attributes of the data that best separate or classify the observations. This process of splitting is done by evaluating which feature provides the greatest distinction between classes or predicts the target variable most effectively (thus providing maximum **information gain**), leading to a clear, branching structure that resembles a tree. Each internal node of the tree represents a decision point based on a particular feature, each branch represents the outcome of that decision, and each leaf node represents a final decision or classification - as shown in Figure 4.6. The simplicity and interpretability of decision trees make them a popular choice for both explanatory purposes and practical applications. They allow users to visualize and understand the decision-making process, making it easy to identify which factors are most influential in determining outcomes. Furthermore, decision trees are versatile, handling both categorical and continuous data, and can be applied to a wide range of tasks, including classification, regression, and feature selection. Despite their straightforward structure, decision trees can model complex decision processes and are often used as the building blocks for more advanced ensemble methods, such as Random Forests and GBM, which further enhance their predictive power and robustness.

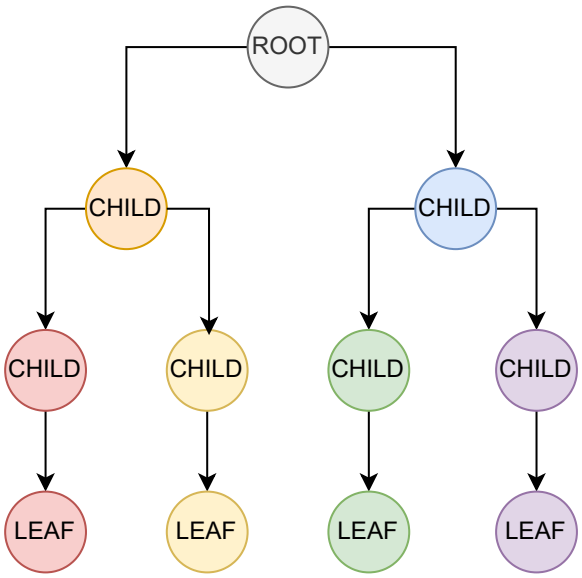

Figure 4.6: Decision Tree architecture based on common designs found in literature.

However, a key challenge with decision trees is the risk of overfitting. As the tree becomes more complex, it may start to capture noise in the data along with the actual signals, leading to a model that performs exceptionally well on the training data but poorly on new, unseen data. To mitigate overfitting, techniques like tree pruning are employed, where the maximum depth of the tree and the number of leaf nodes is limited. This helps in simplifying the model and ensuring it generalizes better to new data. Additionally, in many real-world applications, a variant called Classification and Regression Trees (CART) [18] is used, which provides a score at each leaf node rather than just a binary decision, offering

more nuanced predictions.

Random Forest algorithm, proposed in 2001 [17], has been extremely successful as a general-purpose classification and regression method [12]. Random Forest is a powerful ML algorithm designed to overcome the overfitting problem often encountered in decision trees. It achieves this by ensembling, which involves building multiple weak decision trees and combining their predictions to produce a stronger, more accurate model. In a Random Forest, each tree in the ensemble is trained on a different random sample of the data, and each node in the tree considers only a random subset of features when making splits. This approach ensures that the trees are uncorrelated and diverse, which is crucial for the ensemble to outperform individual trees. This method, known as **bagging**, leverages the idea that a crowd of diverse, weak learners can collectively make more accurate predictions than any single model, reducing the risk of overfitting and improving generalization to new data. For the classification task, the final prediction of the Random Forest is determined by a majority vote across all trees, making the model more robust to errors from individual trees. However, for the regression task, Random Forest uses an averaging method to determine the final prediction - each decision tree in the ensemble provides its numerical prediction, and the final prediction is obtained by averaging these numerical predictions. Random Forest architecture is shown in Figure 4.7.

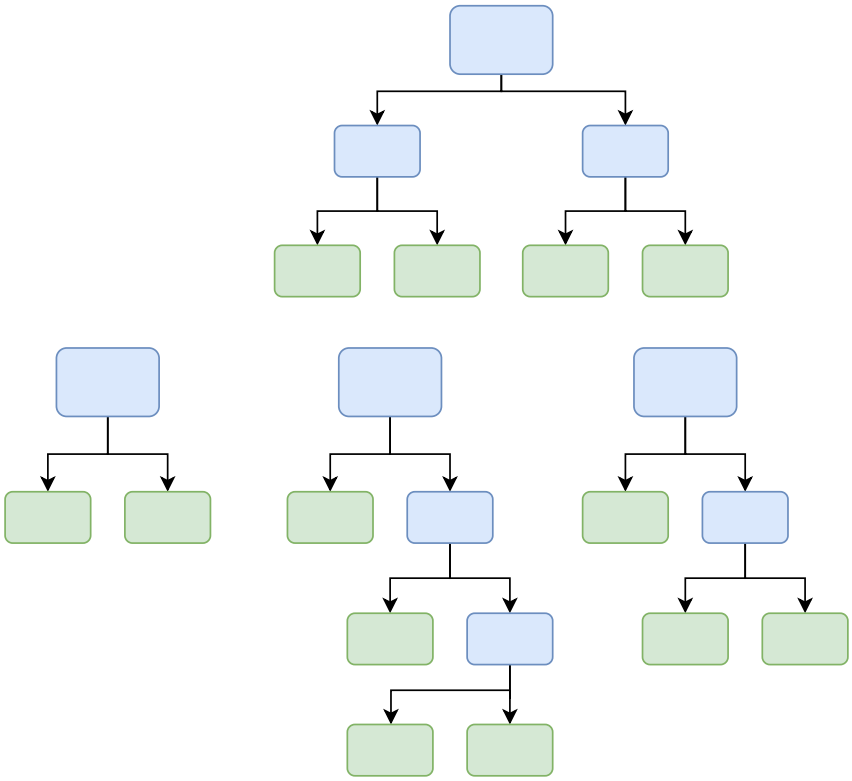

Figure 4.7: Random Forest architecture based on common designs found in literature.

Gradient Boosting Machines [32] were introduced in 2001. [32] builds upon earlier concepts of boosting, which were developed in the 1990s, but Friedman's contribution was the formulation of **boosting** as

a gradient descent algorithm, which allowed for a more general and powerful approach to building ensembles of decision trees. The key idea behind boosting is to build models sequentially, with each new model attempting to correct the errors made by the previous ones. By focusing on the gradient of the loss function concerning the model's predictions, GBM iteratively improves the model's accuracy, making it a highly effective technique for classification and regression tasks. For classification problems, a common loss function is "logarithmic loss" (log loss), which quantifies the accuracy of a classifier by penalizing incorrect classifications. This metric is widely used in many classification challenges to rank model performance. In the case of regression problems, the squared error is typically used as the loss function, where it measures the square of the difference between the predicted and actual values.

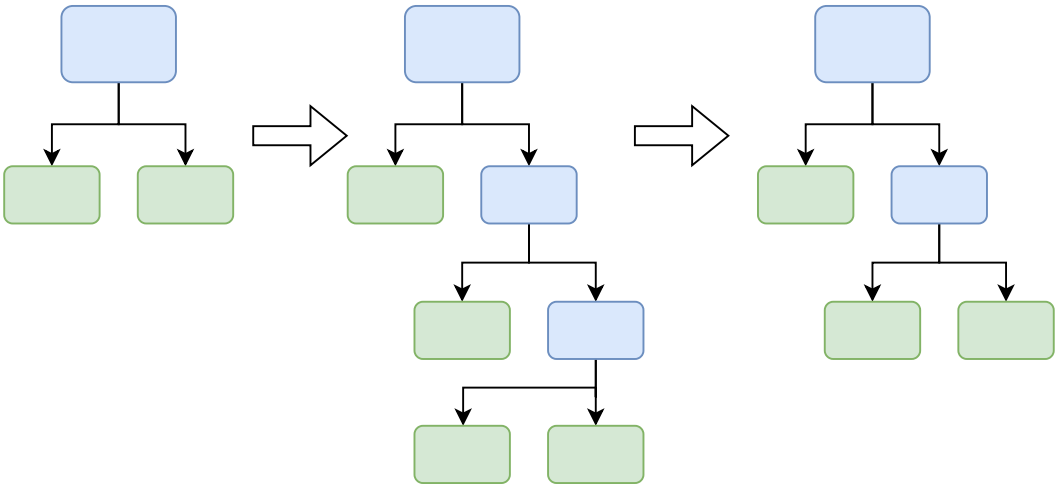

Figure 4.8: Gradient Boosting Machine architecture based on common designs found in literature.

The architecture of the GBM is presented in Figure 4.8. Comparison between boosting and bagging algorithms is also presented in Table 4.2.

Table 4.2: Comparison of Bagging and Boosting Algorithms

Aspect	**Bagging**	**Boosting**
Ensembling Method	Weak learners (decision trees) are ensembled in parallel.	Weak learners (decision trees) are ensembled sequentially.
Learning Process	Learners do not learn from each other.	Learners sequentially minimize the errors of their predecessors.
Influence on Learners	Learners do not boost the influence of high-performing learners.	Learners boost high-performing learners' influence, enhancing the overall model's performance.
Learning Speed	Since the trees are ensembled in parallel, the model learns quickly.	Since the trees are ensembled sequentially, the model learns slowly, but often with better performance.
Dataset Sampling	The dataset used to build new learners consists of samples of the same size as the original dataset, with replacement.	The dataset also consists of samples of the same size with replacement, but data points are weighted, and weights are adjusted after each learner to focus on misclassified points.
Prediction Method	The final prediction is based on the democratic majority vote or the simple average of outputs from all learners.	The final prediction is the weighted average of all learners, with better-performing learners given higher weights.

Chapter 5

Elephant Flow Classification

The concept of managing elephant flows individually dates back to 1999, when Shaikh et al. [102] introduced the idea of selectively routing substantial data flows. Initially, due to the constraints of hardware at the time, these ideas remained largely theoretical and were primarily discussed within academic circles. However, the emergence and advancement of SDN have brought renewed vigor to this approach. SDN enables a centralized controller that possesses an exhaustive view of the network's dynamics to adeptly manage large data flows. This setup allows for traffic engineering strategies that include installing wildcard entries for the shortest paths and actively monitoring traffic to detect elephant flows. Once identified, these large flows can be dynamically rerouted to less congested paths, utilizing the global network view provided by SDN.

Several systems have been developed based on dynamically managing large data flows. Hedera [4] is engineered to redirect significant flows dynamically that surpass predefined thresholds, using flow statistics from OpenFlow counters at edge devices to determine new paths. Similarly, DevoFlow [27] targets elephant flows, employing sampling and threshold techniques for their detection, yet it limits its evaluation to the overall network efficiency. Additionally, Xu et al. [114] have introduced a system that uses a modified version of the Bloom filter to identify elephant flows directly at edge devices.

The studies mentioned primarily utilize basic methods like sampling, counters, and thresholds to detect large data flows. However, there has been a trend towards adopting more advanced, ML techniques in recent years. For example, a study by [111] introduced a decision tree model specifically for detecting elephant flows, highlighting its detection accuracy. Additionally, [90] by Poupart et al. examined three different machine learning approaches for estimating flow sizes and classifying them as elephant flows. They analyzed a dataset of three million flows, including both TCP and UDP protocols, focusing on the effectiveness of these models in correctly identifying large and small flows, measured by the true positive and true negative rates, respectively.

In [71], Liu et al. propose using a random forest decision tree to identify eight key features for a classification model within a dual-layered SDN architecture. This framework begins with an initial sorting phase at the edge devices, followed by a detailed classification at the central controller. They categorize network traffic into four types: elephant, cheetah, tortoise, and porcupine, each with unique traffic characteristics. The system's effectiveness is assessed by the accuracy of the flow classification and the associated time delays. Similarly, Hamdan et al. in [41] present a two-tier classification system for network traffic, starting

with initial filtering at the switches and concluding with detailed classifications at the central controller. They employ the count-min sketch algorithm to differentiate between mice and potential elephant flows at the switch level and a decision tree at the controller for final assessments. The classifier's algorithms are regularly updated based on feedback from the controller, ensuring accuracy with real traffic patterns.

In 2022, He et al. [46] and Qian et al. [93] each introduced sketch-based methods to enhance flow table management. He et al. proposed a single-tier, streamlined approach, whereas Qian et al. developed a TCAM-based system for storing elephant flow labels to improve the balance in identifying both elephant and mouse flows accurately. Both teams conducted evaluations using real packet traces from Internet Service Provider (ISP), demonstrating the real-world applicability and effectiveness of their solutions in managing network traffic.

In their research, da Silva et al. [28] developed a predictive model employing the Locally Weighted Regression (LWR) algorithm to estimate the size and duration of new network flows by analyzing historical flow patterns and their immediate correlations. Later, in 2022, they enhanced this approach by integrating a hashing mechanism inspired by the Cuckoo Search meta-heuristic to improve flow management [19]. Also in 2022, Pekar et al. introduced a threshold-independent system for classifying heavy-hitters [88], which uses template matching to detect elephant flows based on the distribution of packet sizes in the initial packets, providing a refined approach to flow classification without relying on set thresholds.

A hybrid load-balancing strategy that uses Deep Reinforcement Learning (DRL) is outlined in [26]. The so-called CrossBal system is used to manage elephant flows through a multi-stage detection process that includes threshold-based identification and subsequent rerouting for optimal load distribution. Similarly, Wassie et al. [108] implemented a DL strategy utilizing deep autoencoders and advanced ML techniques, including gradient boosting and automated machine learning algorithms like Extreme Gradient Boosting (XGBoost) [24] and GBM [33], to enhance network flow management.

All the referenced studies initially classify network flows by analyzing several of the initial packets. However, the most beneficial is to quickly identify a flow, ideally using only the data available in the first packet. Hardegen et al. [42] introduced a method using a deep neural network for multiclass prediction of flow characteristics from just the first packet's 5-tuple, building on their earlier work [43] where they predicted a flow's bit rate from similar data. Similarly, Durner et al. [29] managed to achieve flow classification effectively by using only the 5-tuple data and the size of the first packet. In more recent research, Gomez et al. [40] in 2023 assessed various ML techniques for classifying network flows based solely on the first packet. Their study, similar to others, concentrated on the accuracy of classification rather than the impact on flow tables or overall traffic coverage. In a 2024 study, Xie et al. [113] introduced a dual-stage decision tree framework specifically for identifying elephant flows. This system initially uses data from the first packet headers and was developed using P4. However, it was tested solely in an emulator; hence it lacks validation on actual hardware. Recent research has leveraged neural networks for network flow classification, focusing on QoS instead of traffic engineering. Alkhalidi et al. [5] developed a one-dimensional CNN that classifies flows into different categories using packet header information, featuring an innovative technique for automatically selecting specific header bits. This method reduces the number of features, processing time, and energy use while achieving satisfactory accuracy. Similarly, Yaseen et al. [115] used a comparable strategy

to classify traffic and set the Differentiated Services Code Point (DSCP) field, integrating their system within an SDN controller and evaluating it using the Mininet emulator for scenarios involving the prioritization of emergency traffic.

The studies mentioned above primarily concentrate on the accuracy of flow classification, such as true positive, and true negative rates, and the accuracy of mouse/elephant binary classification, but overlook the practical effects these algorithms might have on traffic engineering objectives. For instance, the incorrect classification of a network's largest flow could significantly impact traffic coverage, which is more critical than misclassifying smaller flows. The metrics used in these studies fail to capture such nuances. There is a notable absence of emphasis on key traffic engineering metrics, such as the reduction of flow table operations and records or the amount of traffic effectively managed post-classification, which are vital for evaluating the load on network switches and controllers. Subsequent works, however, have shifted focus to these essential metrics, addressing the gaps identified in earlier research. [56] demonstrates that employing a TabNet [7] model can accurately identify elephant flows from the start, reducing flow operations by up to 20 times while managing 80% of the network traffic through individual flow entries. A simple neural network model presented in [66] reduced flow table operations by a factor of 14.7 under the most optimal hyperparameter configuration. Significantly better results for reducing flow operations and flow table records are presented in [51]. Apart from superior results, the authors also explain why traditional ML metrics are inadequate for performance assessment in TE and QoS applications. The authors further refined their work to identify the best model for TE and QoS metrics in [52], where extremely randomized trees were able to reduce flow operations by a factor of 66.35 and the average number of flow table entries by 19.73, while still covering 80% of the traffic.

Identifying the largest flows is a considerable challenge. The objective is to detect these flows early enough to promptly allocate individual entries, ensuring that the majority of their packets follow designated routing paths. **Ideally, flows should be classified from the very first packet to avoid the need for rerouting during the connection.** This method must rely solely on the data contained within packet headers and cannot use information such as packet sizes or interarrival times.

To identify large flows, multiple models were explored using the 5-tuple data from packet headers. A key aspect of the evaluation focused on metrics critical to traffic engineering in SDN. In particular, significant attention was given to two key metrics:

Traffic Coverage: This metric measures the proportion of network traffic (in terms of bytes transmitted) accounted for by flows classified as elephants during the analyzed period. It is calculated by dividing the total bytes transmitted by the predicted elephant flows by the total bytes transmitted across all flows. This metric is expressed in Equation 5.1.

$$\text{TC} = \frac{bytes\_sent\_elephants}{bytes\_sent\_all} \tag{5.1}$$

Flow Operations Reduction: This metric measures the reduction in flow operations resulting from classifying flows as elephants. It is defined as the inverse of the ratio between the number of elephant flows and the total number of flows. In traffic engineering applications, where flow entries are typically created only for

elephant flows, this metric indicates the factor by which the number of flow entry operations (such as creation and deletion) is reduced. This metric is expressed in Equation 5.2.

$$\text{Flow Operations Reduction} = \frac{all\_flows\_number}{elephant\_flows\_number} \tag{5.2}$$

Understanding the inherent tradeoff between these metrics is crucial. Increasing the threshold for detecting elephant flows results in a greater reduction in flow operations but also reduces the proportion of traffic covered. Finding the right balance is essential for optimizing both network efficiency and quality of service (QoS).

The performance of any ML algorithm is heavily dependent on the dataset used. In this evaluation, the data regarding the length and size distributions of flows was collected from a dataset gathered over 30 days in a large campus network [53]. For data processing, the package detailed in [84] was used. The dataset in question contains over 4 billion flows, with the full flow records totaling approximately 278 GB in binary format. Due to its vast size, an anonymized subset of the data was used for training and evaluation of the models, as detailed in [50]. This subset represents one hour of traffic, including 6,517,484 flows and 547 GB of data transmission. This specific period was carefully selected to ensure the absence of anomalies and to closely match the theoretical reduction rate curve of a perfect elephant classifier for this hour with that of the complete 30-day dataset. In the publicly available open-source dataset, IP addresses were anonymized using the prefix-preserving Crypto-PAn algorithm. As shown in [50], this anonymization does not impact the performance of the ML models. This anonymized dataset was partitioned into training and validation sets, comprising 80% and 20% of the data, respectively. This distribution implies that the training sets included 5,213,988 flows, while the validation sets contained 1,303,496 flows.

Input features were derived from the 5-tuple, which consists of the source IP address, destination IP address, source port, destination port, and transport layer protocol, collectively totaling 104 bits. In this research, three distinct representations of the input data were considered:

- **Bits**: each field in the header is decomposed into individual bits, resulting in 104 distinct features, each represented as a binary value (0 or 1).
- **Octets**: fields longer than 8 bits, such as IP addresses and ports, are split into separate octets. This method produces 13 features, with each represented as an 8-bit integer.
- **Raw**: the 5-tuple fields remain unmodified, forming an input vector of five features: source IP, destination IP, source port, destination port, and protocol. Each feature is represented as a 32-bit integer.

Balancing the training dataset played a crucial role in the regression problem. In the training dataset, which contains the previously mentioned 5,213,988 flows, the number of mice flows significantly outnumbers the elephant flows. The numbers express this imbalance in the best possible way: In the fold 0 (first 20%, detailed in Section 5.3.1) for 80% of traffic coverage, the number of elephant flows is 1,253 out of 1,303,497 flows. To address this imbalance, specific balancing techniques were applied; however, this was only relevant to the regression problem and was not used in the classification-based approach. Balancing of the dataset is summarized in the following steps:

1. Set the *ratio* as shown in Equation 5.3,
2. Calculate *balanced dataset size* as shown in Equation 5.4,
3. Sort the initial training dataset in *descending order*,
4. Select the first half of the *balanced dataset* as shown in the Equation 5.5,
5. Select the second half of the *balanced dataset* as shown in Equation 5.6,
6. Join both halves of the balanced dataset as shown in Equation 5.7.

$$ratio = r \tag{5.3}$$

For example, r = 0.10 (i.e., 10%).

$$S = N \times r \tag{5.4}$$

where:

S - size of the balanced dataset,
N - size of the initial training dataset (e.g., 5,213,988),
r - ratio.

$$BD_1 = \mathcal{D}[:, \frac{S}{2}] \tag{5.5}$$

where:

BD_1 - first half of the balanced dataset,
$\mathcal{D}$ - the sorted dataset in descending order,
$\frac{\text{S}}{2}$ - half of the balanced dataset size.

$$BD_2 = RandomSample(\mathcal{D}[\frac{S}{2}, :], \frac{S}{2}) \tag{5.6}$$

where:

BD_2 - second half of the balanced dataset,
$\mathcal{D}$ - the sorted dataset in descending order,
$\frac{\text{S}}{2}$ - half of the balanced dataset size.

$$BD = BD_1 \cup BD_2 \tag{5.7}$$

First, elephant flow prediction was formulated as a regression problem for NN and TabNet models, as described in Sections 5.1 and 5.2. In this approach, the label represents the flow size in bytes. Subsequently, experiments were conducted with elephant flow prediction formulated as a classification problem, as detailed in Section 5.3, where the label is a binary value: 0 for mice flows and 1 for elephant flows. While both NN and TabNet models could also be applied to classification tasks, preliminary results were significantly unsatisfactory compared to the models presented in Section 5.3. Furthermore, Section 5.3 explains why traditional ML metrics fail to meet the criteria for networking QoS and TE applications.

5.1 Neural networks

The performance of several NN models built using *PyTorch* [60] was evaluated. Specifically, six different NN models, five distinct label normalization approaches, and a single input data representation (*bits*) were tested.

5.1.1 Training

The training phase consisted of three key steps: balancing of the dataset, hyperparameter selection, label normalization, and NN training. A flowchart illustrating this process is presented in Figure 5.1.

Each NN model was trained on a *shuffled, balanced dataset*. Both training and validation were conducted using multiple *hyperparameter* combinations. The variable and constant hyperparameters are listed in Table 5.1 and Table 5.2, respectively, with the constant hyperparameters remaining unchanged across all training sessions.

Table 5.1: Neural networks variable hyperparameters

Batch Size	**Learning Rate**	**Balancing ratio**
1280	$1e^{-3}$	6%
2560	$3e^{-3}$	10%
5120	$6e^{-3}$	20%

Table 5.2: Neural networks constant hyperparameters

Epochs	**Optimizer**	**Loss Function**
100	Adam	Mean Absolute Error

In general, a larger *batch size* leads to more stable training and reduces the risk of overfitting. To account for this, the learning rate was adjusted alongside the batch size. This adjustment enabled the model to locate local minima and maxima more efficiently without requiring a proportional increase in the number of epochs.

The selection process for the *balancing dataset ratios* required careful consideration to ensure a sufficient number of elephant flows were included while effectively managing computational resources.

- **6% ratio** - a conservative option, balancing the inclusion of elephant flows with the need to maintain manageable computational demands. This ratio ensures that the model can be trained without overwhelming resources while still providing an adequate representation of elephant flows.

- **10% ratio** - a more substantial inclusion of elephant flows, offering a broader representation in the training data. This ratio strikes a middle ground, providing a more inclusive dataset while requiring additional computational power.

- **20% ratio** - an even more comprehensive inclusion of elephant flows, offering the widest representation among the selected ratios. However, this ratio also demands the highest computational resources.

Preliminary research showed that a 100% balancing ratio led to completely unsatisfactory results—none of the models were able to predict elephant flows due to the overwhelming numerical dominance of mice flows in the dataset. Therefore, this ratio is not considered further for NN models.

Constant parameters that remained constant for all training sessions were: the number of epochs, optimizer, and, loss function. The models were trained for 100 epochs to ensure sufficient learning time for convergence. This duration was selected based on preliminary experiments indicating that 100 epochs allowed the models to adequately learn from the data without overfitting. Mean Absolute Error (MAE) was selected as a loss function for trained models. **MAE** is expressed by Equation 5.8. This metric is calculated by taking the average of the absolute differences between the predicted values and the actual values. It measures the average magnitude of errors in a set of predictions, without considering their direction. MAE is less sensitive to outliers because it treats all errors equally by taking the absolute value. This means that a large error has the same impact on the MAE as a small error, making it a useful metric when you want to assess the average error magnitude without heavily penalizing larger deviations.

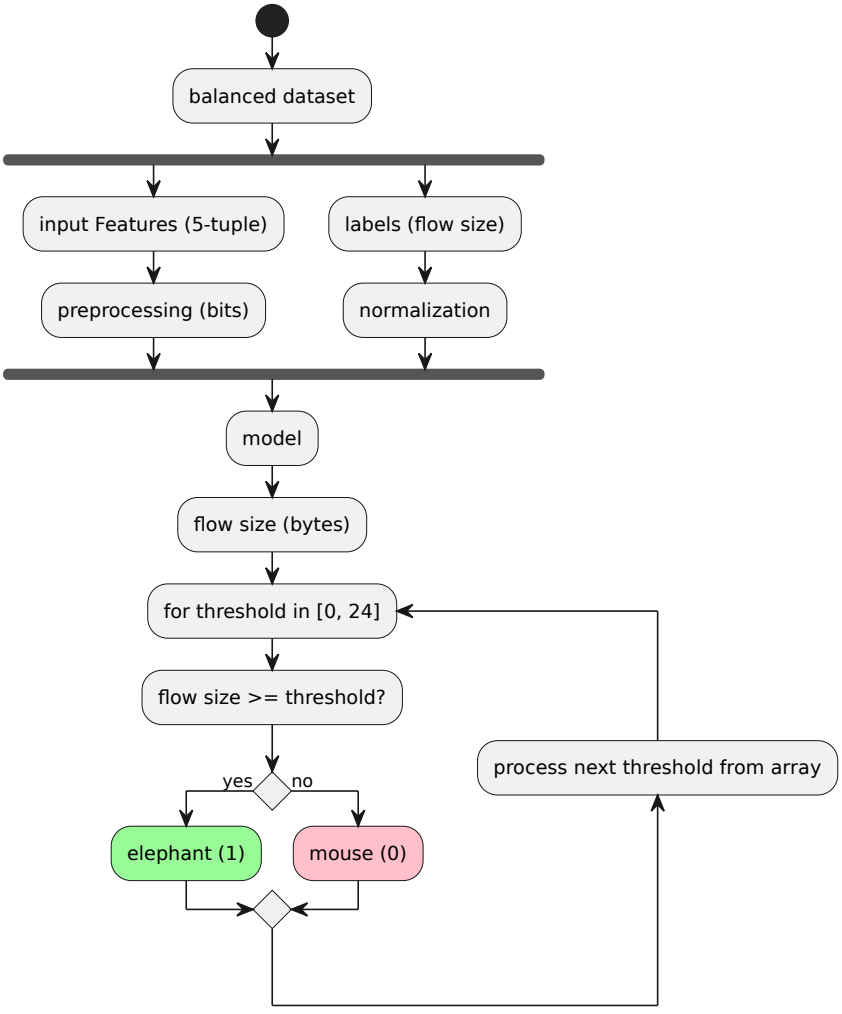

Figure 5.1: Training of the neural network.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{5.8}$$

where:

n - number of data points,

y_i - the actual value of the i-th data point,

$\hat{y}_i$ - predicted value for the i-th data point.

Adaptive Moment Estimation (Adam) [62] was used as the optimizer during all training sessions. An **optimizer** is a critical component that guides the learning process by iteratively adjusting the model's parameters to minimize the loss function. The choice of optimizer significantly impacts both the training speed and the model's overall performance. Adam's ability to efficiently manage large datasets and complex models has made it the optimizer of choice for many state-of-the-art research. Its adaptive learning rate capabilities, which enhance convergence speed and stability, were key factors in its selection for this research.

Every training and validation was performed with different types of *label normalization*. In total, 5 different approaches were examined. Those normalization types are referred to as *NONE*, *ONE CLIP*, *ONE NO CLIP*, *SQUARED CLIP*, and *SQUARED NO CLIP*. The concepts behind those types are presented below.

- **NONE** - there is no normalization of the labels. Original labels are used to train and evaluate the model. Original labels range from 64 to 3,218,210,994 bytes - the smallest, and the biggest flow size.
- **ONE CLIP** - every label from the dataset is divided by one gigabyte (1,000,000,000). After division values from the datasets are clipped to range 0 – 1. This is expressed by the Equation 5.9.
- **ONE NO CLIP** - every label from the dataset is divided by one gigabyte. This is expressed by the Equation 5.10.
- **SQUARED CLIP** - every label from the dataset is squared and divided by one gigabyte. The result is clipped to the range 0 – 1. This is expressed by the Equation 5.11,
- **SQUARED NO CLIP** - every label from the dataset is squared and divided by one gigabyte. This is expressed by the Equation 5.12.

Let labels $= \{l_1, l_2, \ldots, l_n\}$, then for each label l_i in labels, the normalized value $T(l_i)$ is defined by:

$$T(l_i) = \begin{cases} 0 & \text{if } \frac{l_i}{1000000000} < 0, \\ 1 & \text{if } \frac{l_i}{1000000000} > 1, \\ \frac{l_i}{1000000000} & \text{otherwise.} \end{cases} \tag{5.9}$$

$$T(l_i) = \frac{l_i}{1000000000} \tag{5.10}$$

$$T(l_i) = \begin{cases} 0 & \text{if } \frac{l_i^2}{1000000000} < 0, \\ 1 & \text{if } \frac{l_i^2}{1000000000} > 1, \\ \frac{l_i^2}{1000000000} & \text{otherwise} \end{cases} \tag{5.11}$$

$$T(l_i) = \frac{l_i^2}{1000000000} \tag{5.12}$$

5.1.2 Models

The primary difference between the examined models lies in the number of layers and the number of neurons. However, they share the same fundamental structure — all models are built using Linear layers and Rectified Linear Activation Function (ReLU) activation functions. Additionally, all models are fully connected feedforward models, with the input and output layers containing the same number of neurons. As previously mentioned, the input layer consists of 104 features, and the output layer represents the predicted flow size, which is a single output. The names of the models are presented in Table 5.3.

Table 5.3: Model names

Model	Name
1	FirstRegression
2	FallingRegression
3	GrowingRegression
4	ThreeLayerNeuralNetworkRegression
5	FourLayerNeuralNetworkRegression
6	TenLayerNeuralNetworkRegression

Figure 5.2 illustrates the architecture of FirstRegression, a neural network with 104 neurons in the input layer, a single hidden layer containing 1024 neurons, and an output layer with one neuron. The architecture of FallingRegression is shown in Figure 5.3. This model has three hidden layers: the first with 1024 neurons, the second with 256 neurons, and the third with 128 neurons. GrowingRegression, depicted in Figure 5.4, is the reverse of FallingRegression. It starts with 128 neurons in the first hidden layer, followed by 256 neurons in the second layer, and 1024 neurons in the third layer. ThreeLayerNeuralNetworkRegression, FourLayerNeuralNetworkRegression, and TenLayerNeuralNetworkRegression all feature hidden layers with 1024 neurons each. The key difference among these models is the number of hidden layers. Their architectures are shown in Figures 5.5, 5.6, and 5.7, respectively.

5.1.3 Results

The relationship between flow operations reduction and traffic coverage can be visualized by plotting a curve, which is generated by simulating decision-making processes where the elephant flow threshold is adjusted according to the predicted flow sizes. The following graphs illustrate how flow operations change as traffic coverage varies. On the y-axis, the unit represents a multiplier (hence [x]). For example, a value

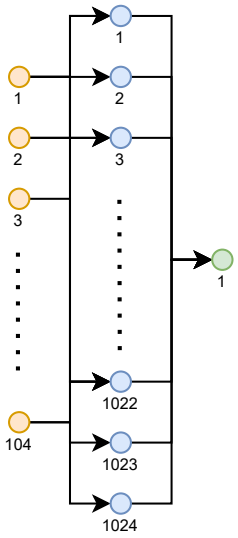

Figure 5.2: FirstRegression

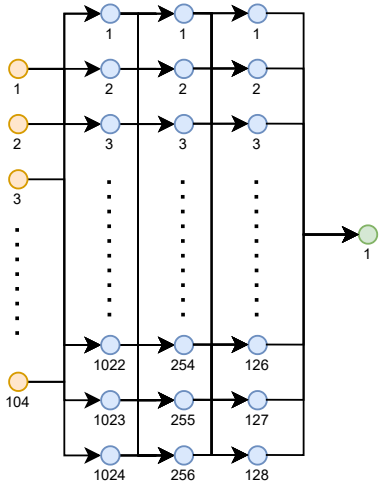

Figure 5.3: FallingRegression

of 1000 means a reduction by a factor of 1000, so the flow operations are reduced to 1/1000 of their initial value. The goal is to reduce flow operations while maintaining optimal traffic coverage. The closer the curve approaches the top-right corner of the graph, the more effective the model is. The **Data** black line represents the projected performance based on a distribution fitted to the validation dataset. This projection assumes a perfect prediction of each flow's size from its first packet. The benchmark for evaluation is the theoretical flow operations reduction rate curves of a model that perfectly predicts the size of all flows from their first packet. This method, described in [49], is referred to as the *first* approach. It involves selecting the smallest number of largest flows, ordered by size in descending order, which together account for a predetermined percentage of total network traffic. Figures 5.8, 5.9, and 5.10 display the results for normalization with clipping, while Figures 5.11, 5.12, and 5.13 present the results for normalization without clipping. In each figure, the best model and training epoch are selected based on the optimal combination of hyperparameters. For example, the appearance of the TenLayerNeuralNetworkRegression_29_NONE curve

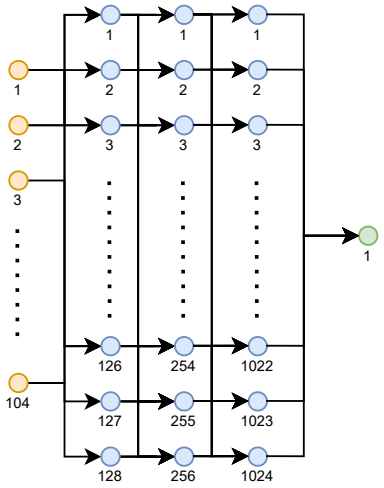

Figure 5.4: GrowingRegression

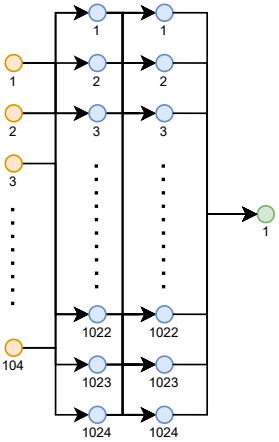

Figure 5.5: ThreeLayerNeuralNetworkRegression

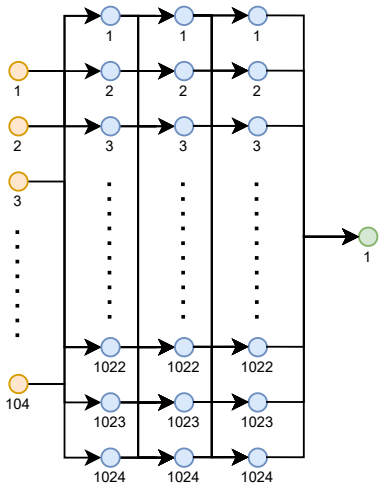

Figure 5.6: FourLayerNeuralNetworkRegression

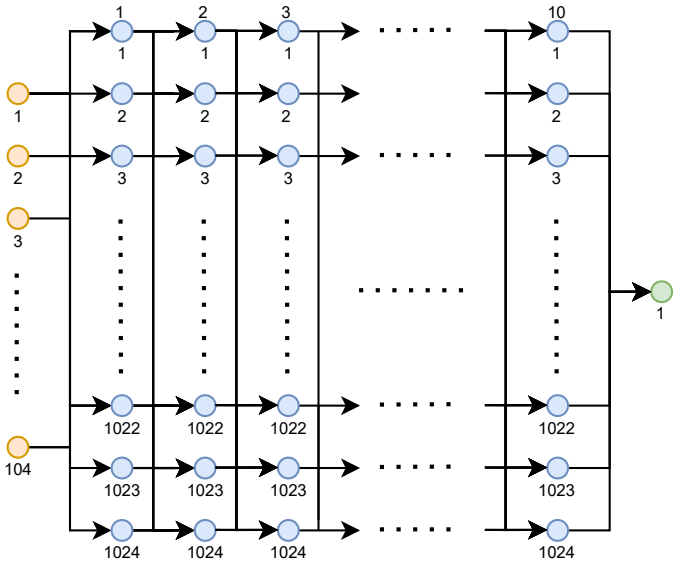

Figure 5.7: TenLayerNeuralNetworkRegression

indicates that, with *NONE* normalization, the TenLayerNeuralNetworkRegression model achieved its best performance at epoch 29. As shown, in some cases, the results do not reach 50% traffic coverage. This is particularly noticeable with the *SQUARED* normalization, where the model predicts negative values, which, after renormalization, are converted to zeros. As a result, the data points produced by the flow operations reduction algorithm often fail to achieve 50% traffic coverage.

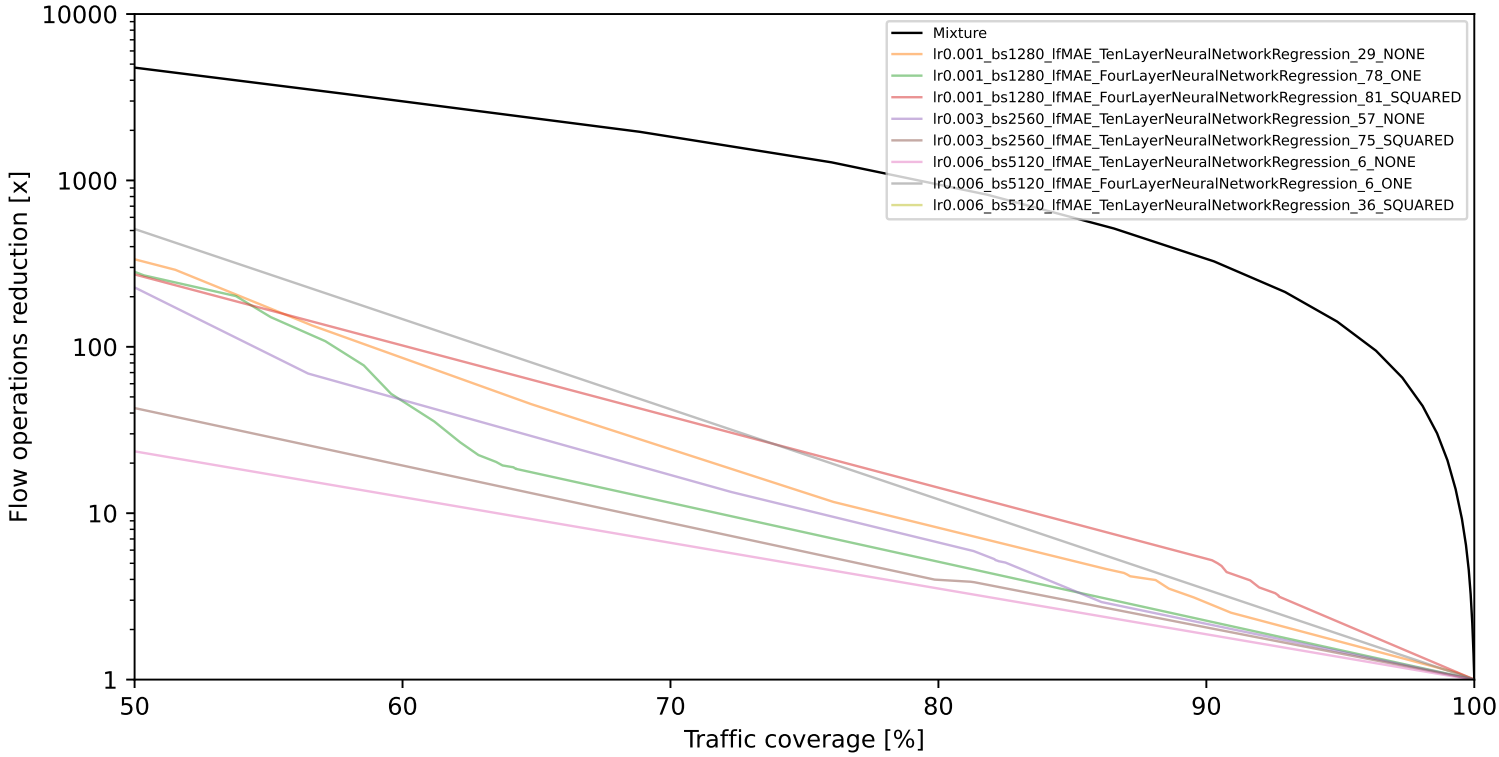

Figure 5.8: Flow operations reduction for the balanced dataset with 6% ratio and *clipping* in the normalization.

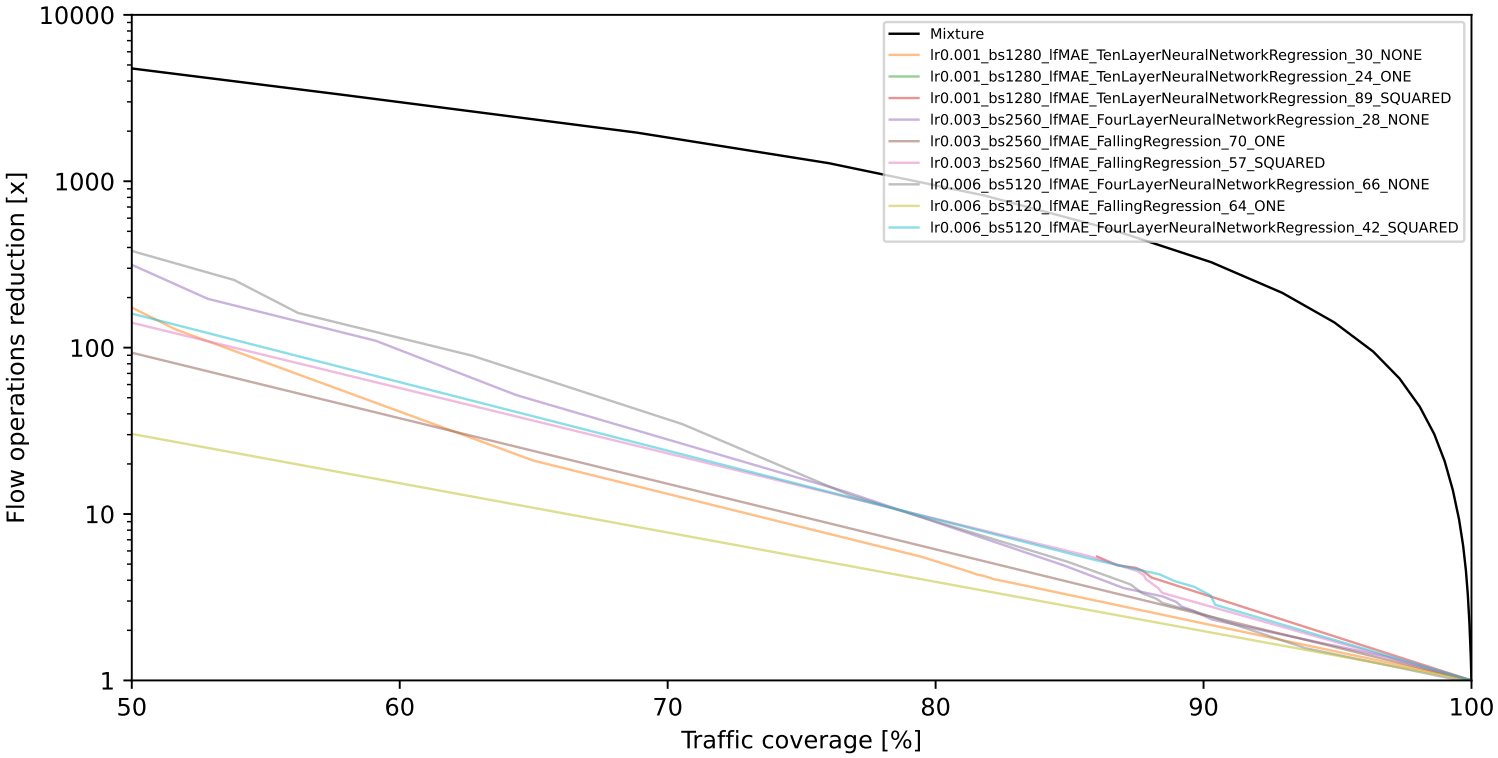

Figure 5.9: Flow operations reduction for the balanced dataset with 10% ratio and *clipping* in the normalization.

Table 5.4: Top performing NN models

Dataset ratio	Model	Normalization	Batch size	Learning rate	Epoch	Flow operations reduction	Traffic coverage
6%	FourLayerNeuralNetworkRegression	SQUARED CLIP	1280	0.001	81	**14.7**	80%
10%	FallingRegression	SQUARED CLIP	2560	0.003	57	9.1	80%
20%	FourLayerNeuralNetworkRegression	ONE CLIP	1280	0.001	87	9.3	80%
6%	FourLayerNeuralNetworkRegression	NONE	1280	0.001	73	11.7	80%
10%	FourLayerNeuralNetworkRegression	NONE	2560	0.003	61	10.8	80%
20%	FallingRegression	ONE NO CLIP	2560	0.003	24	9.0	80%

For each dataset ratio, the best model and training epoch were selected based on the optimal combination of hyperparameters, as shown in Table 5.4. Different normalization techniques appear to have a significant impact on the flow operations reduction metric, with *SQUARED CLIP* generally resulting in higher reduction percentages, particularly when used with the FourLayerNeuralNetworkRegression model at a 6% dataset ratio. Notably, increasing the dataset ratio from 6% to 20% does not lead to a proportional improvement in crucial metrics, indicating that merely increasing the amount of training data is not sufficient to enhance model performance. By utilizing relatively simple NNs composed solely of linear layers to detect elephant flows from the first packets, it is possible to achieve a roughly 15-fold reduction in the number of flow operations while still covering 80% of the traffic.

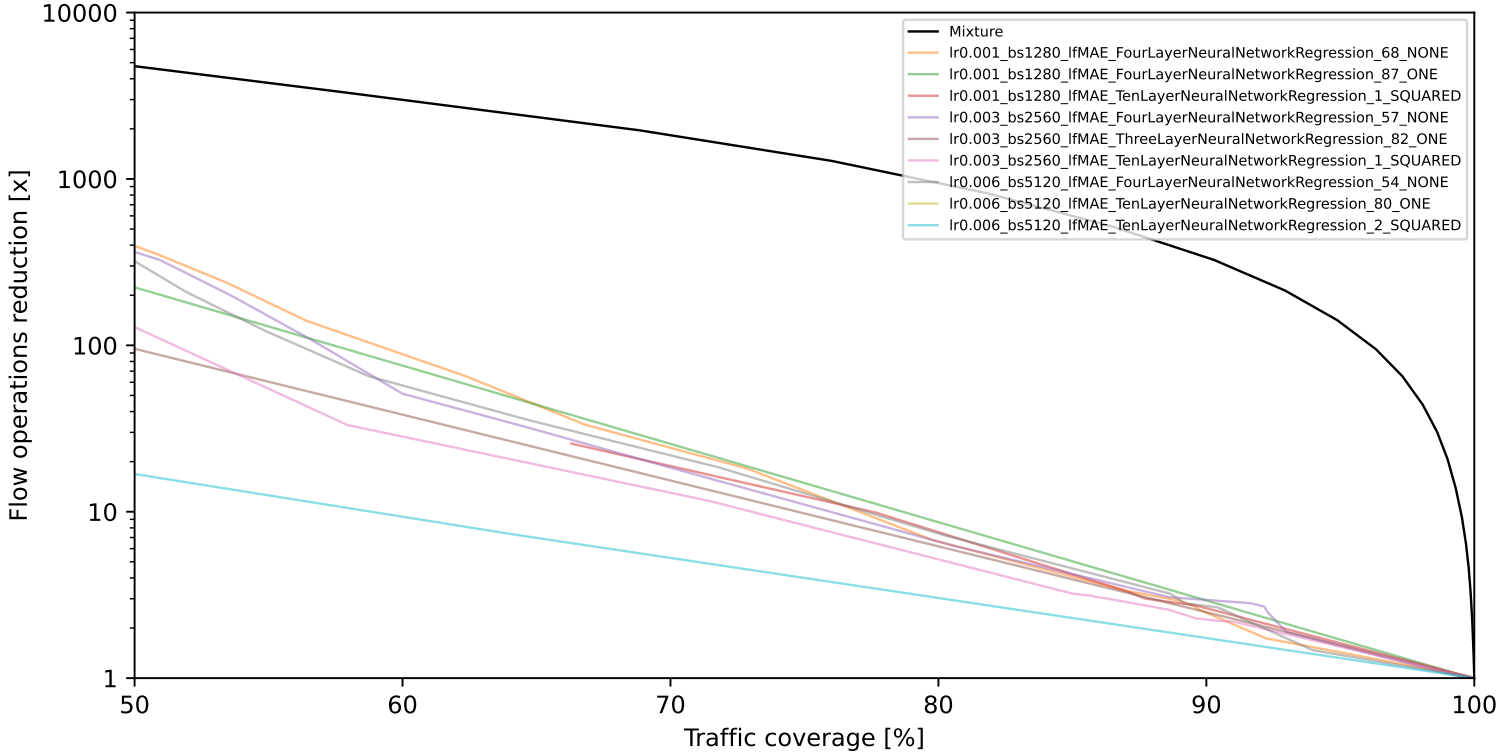

Figure 5.10: Flow operations reduction for the balanced dataset with 20% ratio and *clipping* in the normalization.

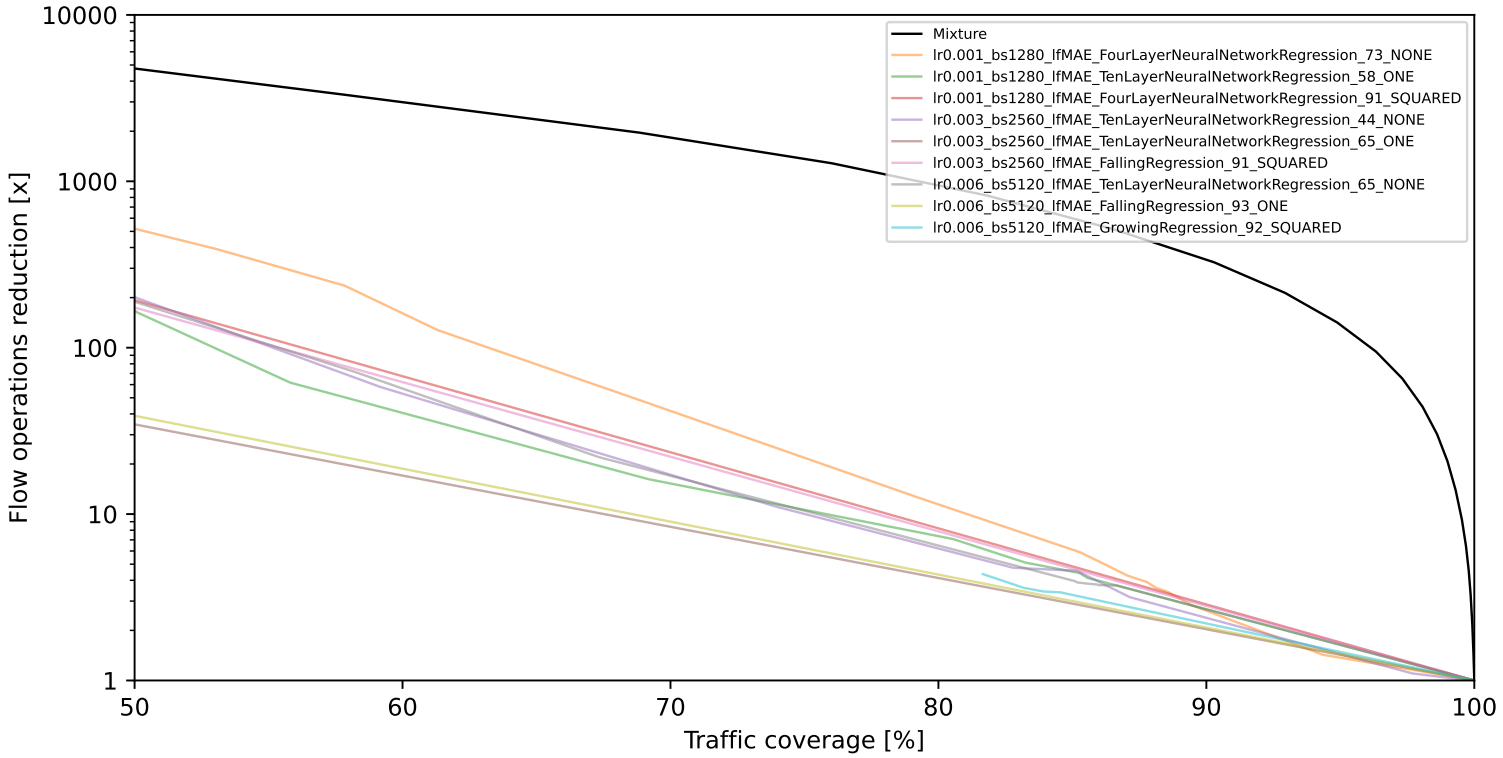

Figure 5.11: Flow operations reduction for the balanced dataset with 6% ratio and *no clipping* in the normalization.

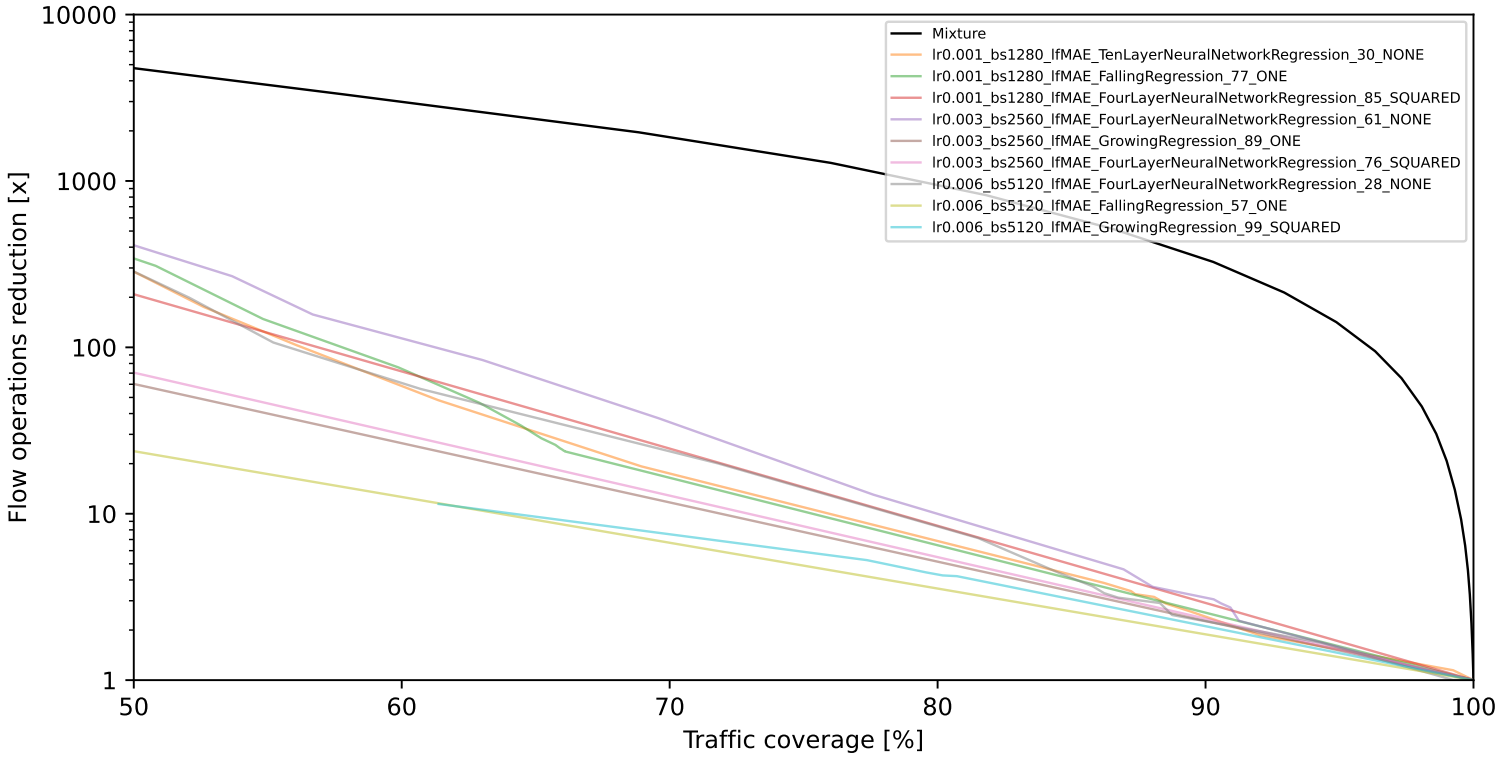

Figure 5.12: Flow operations reduction for the balanced dataset with 10% ratio and *no clipping* in the normalization.

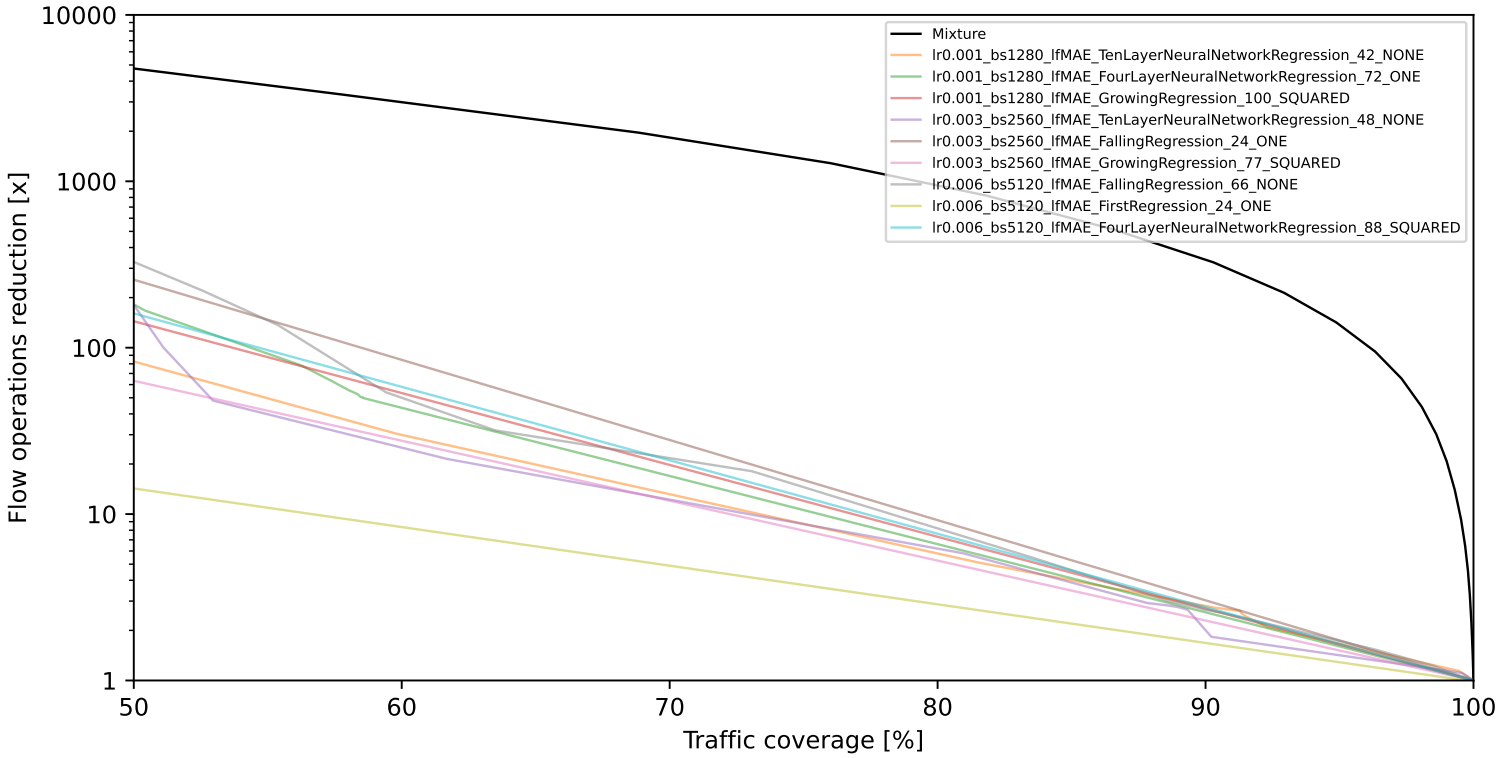

Figure 5.13: Flow operations reduction for the balanced dataset with 20% ratio and *no clipping* in the normalization.

5.2 TabNet

TabNet [7] is a neural network architecture specifically designed for handling tabular data. Developed by researchers at Google Cloud AI, TabNet employs sequential attention mechanisms to selectively determine which features to process at each decision step. This approach enables the model to focus on important, learned features, similar to how decision trees isolate key attributes, but with the enhanced flexibility and power of a neural network. One of TabNet's key strengths is its ability to promote interpretable decision-making, making it particularly valuable in applications where transparency in how input features influence predictions is crucial. TabNet has demonstrated competitive performance on various benchmark datasets, often outperforming traditional ensemble models like random forests and gradient boosting machines (GBM) in certain cases.

The TabNet[1] model was employed to solve the regression problem, utilizing two distinct label normalization approaches and two input data representations (*bits* and *octets*).

In addition to the key metrics (*flow operations reduction* and *traffic coverage*), the following analyses were conducted with TabNet:

- **Entropy Analysis:** examines the average information entropy contained in each 5-tuple field in the packet headers.
- **Feature Significance:** the most significant 5-tuple fields for predictions are identified, providing insight into their contribution to model performance.

5.2.1 Training

The training phase consisted of the following steps: balancing of the dataset, hyperparameters selection, label normalization, and the training of the TabNet. The flowchart of this phase is presented in Figure 5.14.

The model was trained on a *shuffled, balanced dataset*, and its performance was evaluated using a validation dataset. During the training and validation processes, various combinations of hyperparameters were tested. The hyperparameters that were adjusted throughout the training sessions are outlined in Table 5.5, while those that remained consistent across all training sessions are listed in Table 5.6. Additionally, Table 5.7 provides details on the parameters specific to the TabNet model that were varied during the training.

Table 5.5: TabNet variable hyperparameters

Batch Size	Learning Rate	Loss Function	Balancing ratio
2560	$1e^{-3}$	MAE	10%
5120	$3e^{-3}$	Mean Square Error (MSE)	20%
10240	$6e^{-3}$	-	100% (no balancing)

Varying the *batch size* was crucial for observing its impact on model convergence and generalization. The chosen range of 2560 to 10240 was selected for several reasons:

[1] `https://github.com/dreamquark-ai/tabnet`

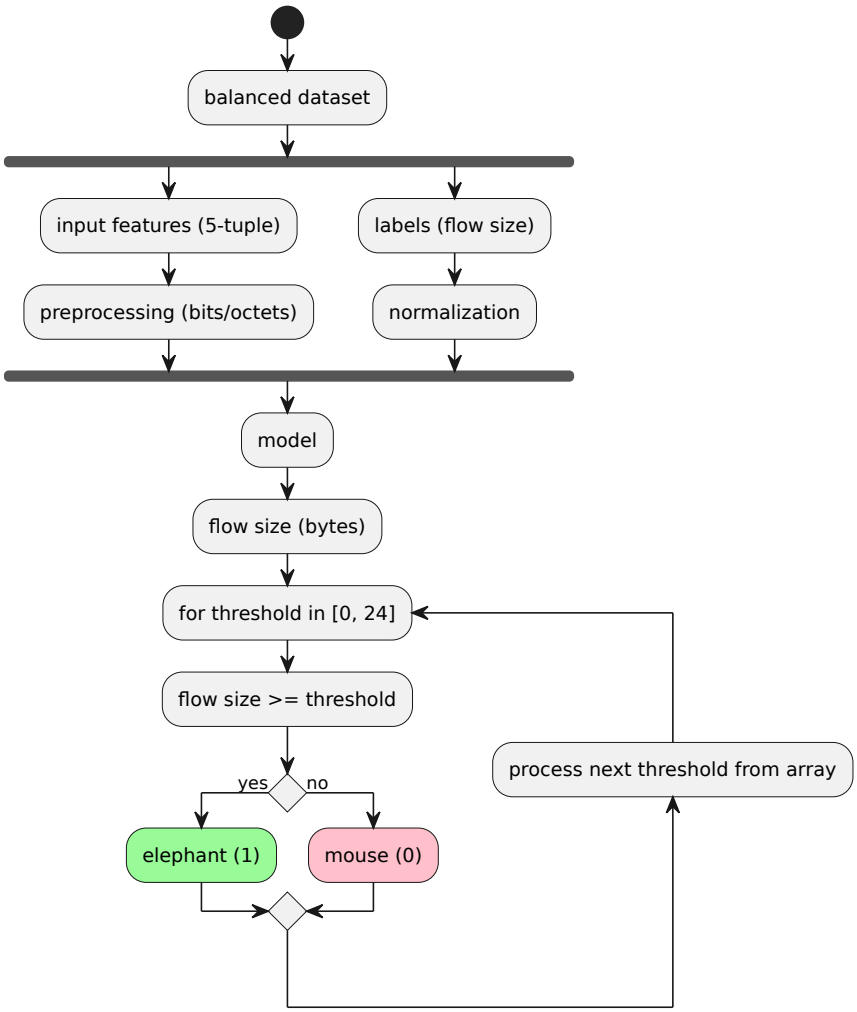

Figure 5.14: Training of the TabNet.

- **Computational Efficiency:** Batch sizes within this range were chosen to strike a balance between memory usage and computational efficiency. Smaller batch sizes could lead to inefficient Graphics Processing Unit (GPU) utilization, while excessively large batch sizes might exceed hardware memory limits, causing slower training times due to increased paging or the need to reduce model complexity,

- **Empirical Performance:** Preliminary experiments demonstrated that this range of batch sizes produced strong performance across various metrics. A batch size of 2560 offered a good balance between frequent weight updates and manageable noise in gradient estimates. Larger batch sizes, such as 5120 and 10240, provided more stable training with slightly slower convergence, which helped in achieving better generalization on the validation set.

- **Model and Data Characteristics:** The characteristics of the dataset and the complexity of the TabNet architecture also influenced the choice of batch size. Given the training dataset size (5,213,988 flows) and the demands of the TabNet architecture, batch sizes within this range were effective in leveraging the computational power of modern GPUs while maintaining efficient training dynamics.

Different *learning rates* were tested to strike an optimal balance between convergence speed and training stability. A lower learning rate of $1e^{-3}$ was used to allow for finer weight adjustments, reducing the risk of overshooting the minima and promoting more stable training. In contrast, a higher learning rate of $6e^{-3}$ was employed to accelerate convergence, though it required careful tuning to prevent potential instability during the training process.

Different *loss functions* were used to distinguish the hyperparameters that varied between the NN models described in Section 5.1.1. MAE and MSE are both commonly used metrics to evaluate the performance of regression models, but they differ in how they handle errors. **MSE**, as expressed in Equation 5.13, is calculated by averaging the squared differences between the predicted and actual values. By squaring the errors, MSE places greater emphasis on larger discrepancies, making it more sensitive to outliers. A single large error can disproportionately increase the MSE, which makes this metric particularly useful when larger errors need to be penalized more heavily.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.13}$$

where:

n - number of data points,

y_i - the actual value of the i-th data point,

$\hat{y}_i$ - predicted value for the i-th data point.

The last variable hyperparameter was the *balancing dataset ratio*. A conservative ratio of 10% was chosen, while a more substantial inclusion of elephant flows was achieved with a 20% ratio. The final selected ratio was 100% (using the dataset as-is), which served as a control to compare the effects of balancing against the unbalanced data.

Table 5.6: TabNet constant hyperparameters

Epochs	**Optimizer**
200	Adam

The model was trained for up to 200 epochs to allow sufficient time for convergence. This duration was based on preliminary experiments, which indicated that 200 epochs were enough for the model to effectively learn from the data without overfitting. However, training did not always run for the full 200 epochs, as an *early stopping* feature was implemented to halt training when no significant performance improvement was observed over a specified number of epochs.

Varying the width of the decision layer (8, 16, 32) allows for the exploration of how model capacity affects performance. A wider layer has the potential to capture more complex patterns, though it also increases the risk of overfitting. Similarly, adjusting the width of the attention embedding (8 , 16, 32) enables an examination of how the model's attention mechanism scales with complexity. Wider embeddings can capture

Table 5.7: TabNet parameters

Width of the decision prediction layer	Width of the attention embedding for each mask	Number of steps in the architecture
8	8	3
16	16	6
32	32	9

more intricate feature interactions. Additionally, the number of steps (3, 6, 9) dictates how many sequential decision and attention layers the data traverses. Increasing the number of steps can enhance model performance by enabling more complex transformations, though this comes with higher computational demands.

Every training and validation was performed with two types of label normalization methods denoted as *NONE* and *MINMAX*:

- **NONE** - there is no normalization of the labels. Original labels are used to train and evaluate the model. Original labels range from 64 to 3,218,210,994 bytes - the smallest, and the biggest flow size.
- **MINMAX** - minimum value from the dataset becomes 0, maximum value from the dataset becomes 1. All other values are scaled proportionally to fall within the range of 0 to 1. The procedure is detailed in Equation 5.14.

Let labels $= \{l_1, l_2, \ldots, l_n\}$, then for each label l_i in labels, the normalized value $T(l_i)$ is defined by:

$$T(l_i) = \frac{l_i - l_{min}}{l_{max} - l_{min}} \tag{5.14}$$

5.2.2 Results

From more than 500 unique results only the best results per input data type and dataset ratio were selected for presentation. In this context, the best means the model which for the 80% traffic coverage produces the biggest flow operations reduction. The graphs below present how flow operations change with traffic coverage with the y-axis having a logarithmic scale.

Figures 5.15, 5.16, and 5.17 display results for the bits input data representation with balanced datasets at ratios of 10%, 20%, and 100%, respectively. In contrast, Figures 5.18, 5.19, and 5.20 present results for the octets input data representation with the same dataset ratios. Additionally, as shown in Figures 5.17 and 5.20, it was not possible to plot the reduction versus coverage results for the MAE with *MINMAX* normalization, as the results did not fall within the traffic coverage area of interest (50-100%). This is because the model in these configurations was significantly underfitted and unable to effectively recognize trends and patterns from the input data. The five best models are summarized in Table 5.8. Additionally, in the last column, a standard deviation was added.

The results of the *flow operations reduction* analysis highlight the superiority of the MSE loss function over MAE. The key advantage of MSE lies in its error amplification mechanism, where larger errors are squared, thereby magnifying their impact and accelerating the minimization process during training. This feature allows MSE to better handle outliers, which seems to be particularly beneficial in this scenario. Additionally, the results indicate that TabNet performed significantly better with unnormalized labels

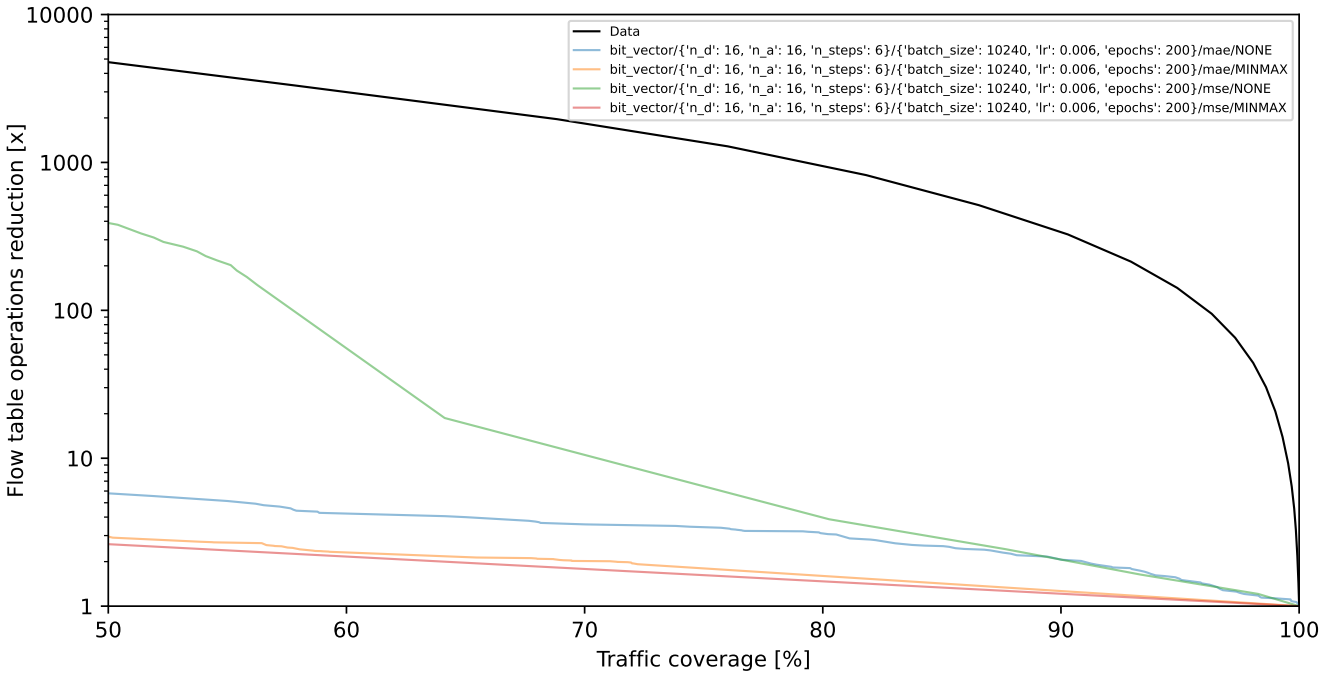

Figure 5.15: Flow operations reduction for the balanced dataset with *10% ratio* and *bits* input data.

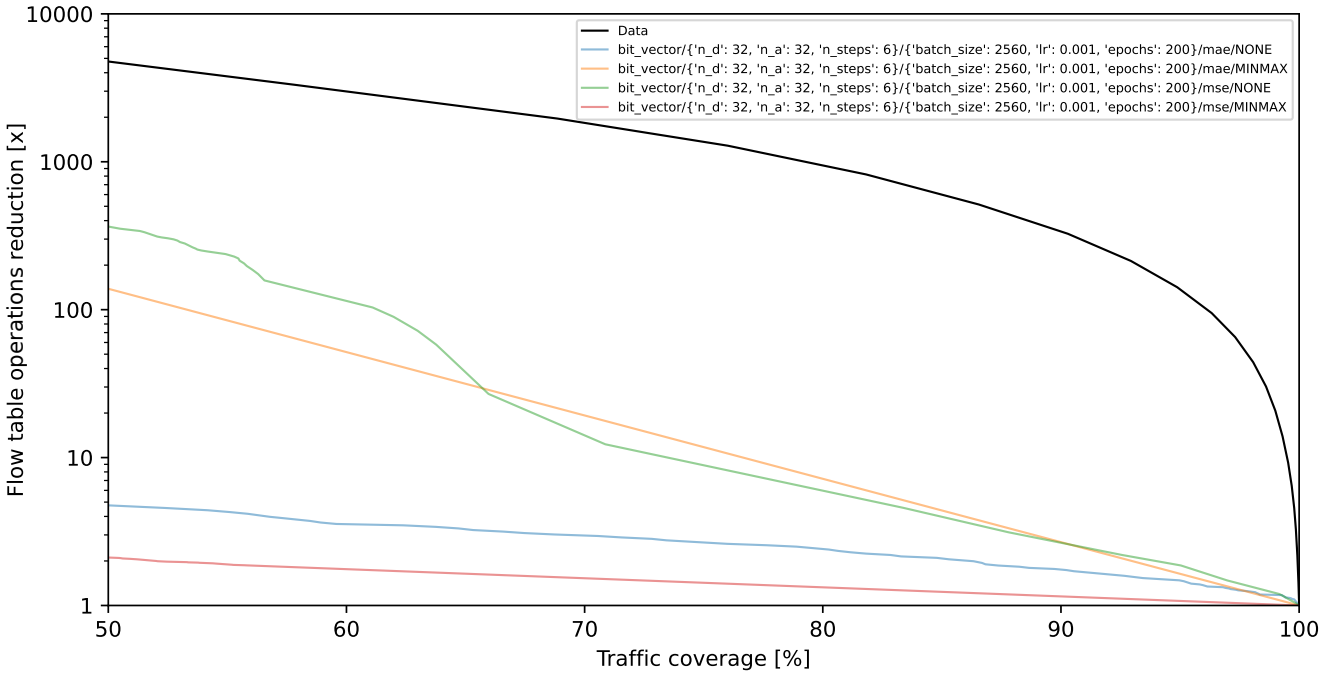

Figure 5.16: Flow operations reduction for the balanced dataset with *20% ratio* and *bits* input data.

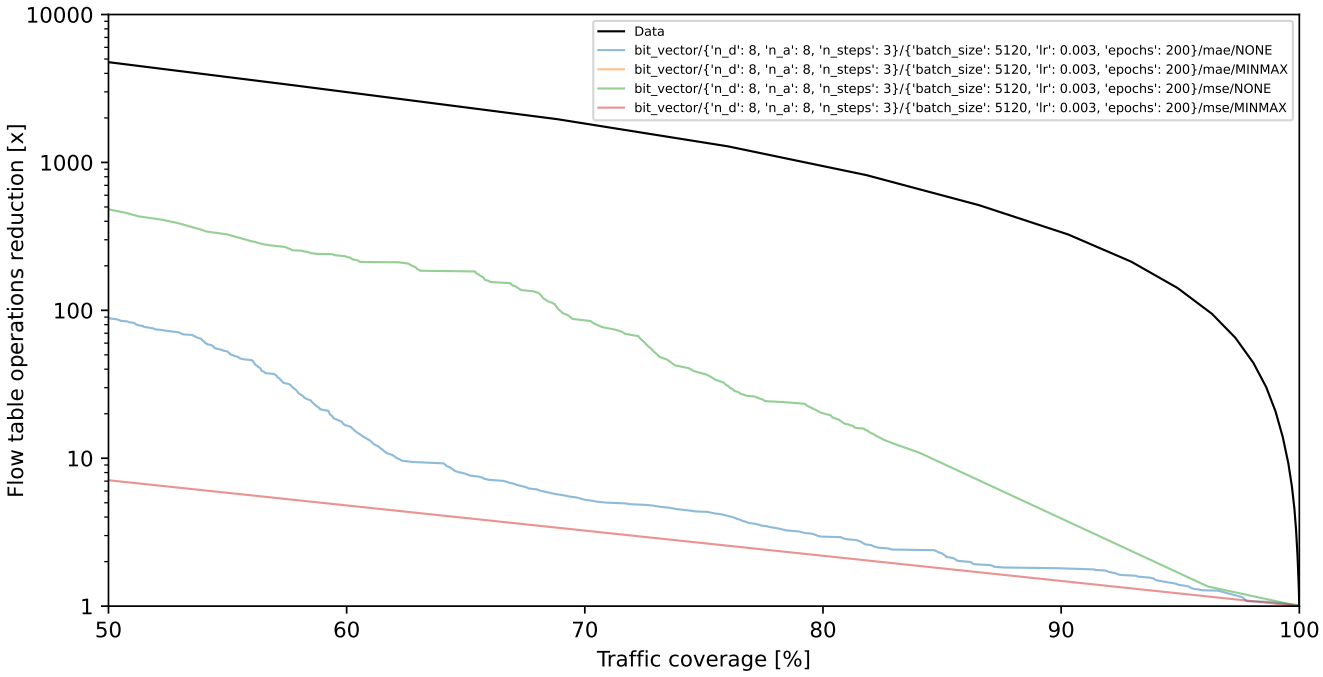

Figure 5.17: Flow operations reduction for the balanced dataset with *100% ratio* and *bits* input data.

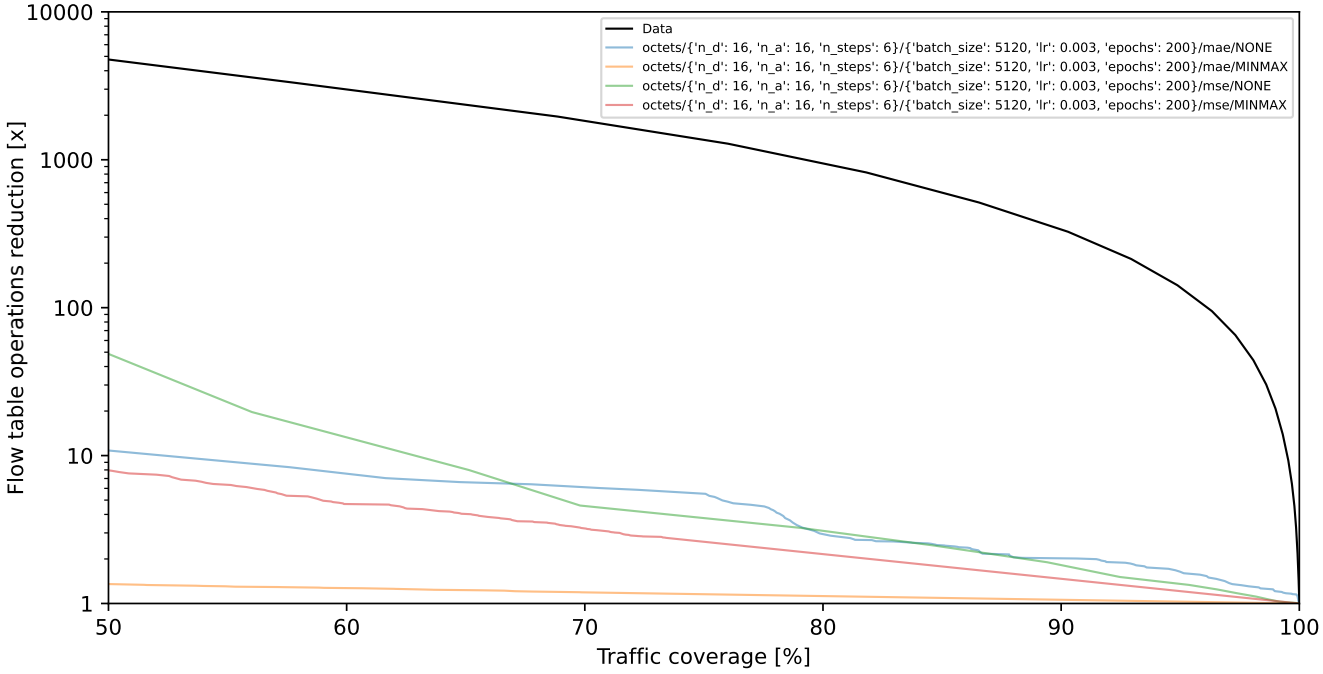

Figure 5.18: Flow operations reduction for the balanced dataset with *10% ratio* and *octets* input data.

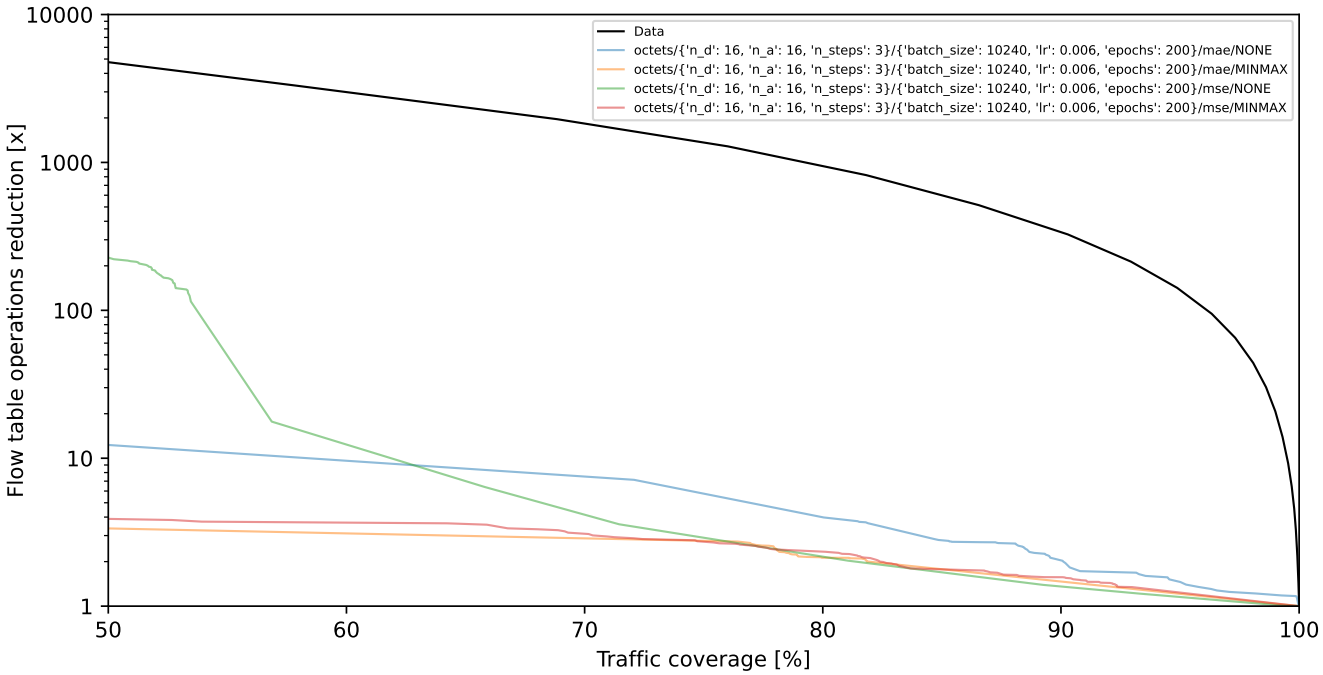

Figure 5.19: Flow operations reduction for the balanced dataset with *20% ratio* and *octets* input data.

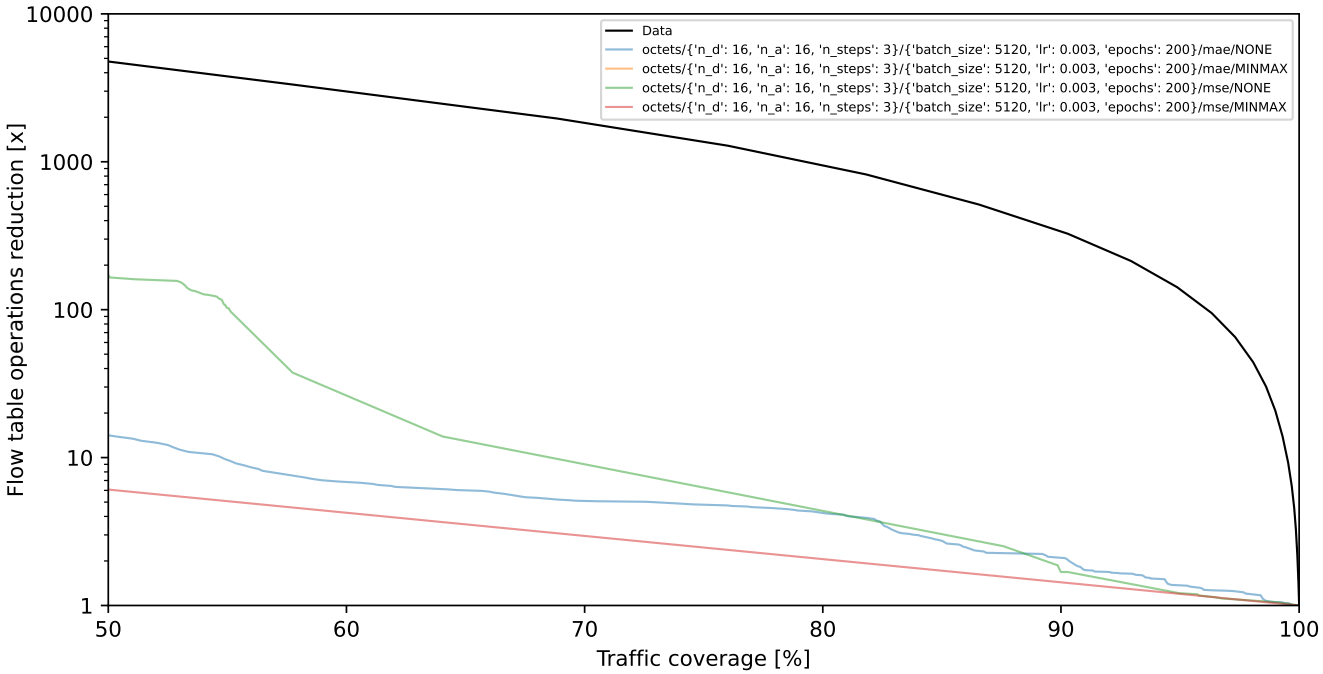

Figure 5.20: Flow operations reduction for the balanced dataset with *100% ratio* and *octets* input data.

Table 5.8: Top performing TabNet models

Input data	Dataset ratio	Model parameters	Normalization	Loss function	Batch size	Learning rate	Flow operations reduction (best)	Flow operations reduction (SD)
bits	100%	8/8/3	NONE	MSE	5120	0.003	**20.14**	5.32
bits	100%	16/16/3	NONE	MSE	2560	0.001	18.74	3.79
bits	100%	8/8/3	NONE	MSE	2560	0.001	14.25	2.98
bits	10%	32/32/3	NONE	MSE	10240	0.006	12.92	2.61
bits	20%	16/16/3	NONE	MSE	2560	0.001	12.59	2.51

compared to normalized labels. Among the various TabNet configurations, the optimal performance was achieved when the width (both for the decision prediction layer and the attention embedding for each mask) was set to 8 and the number of layers was set to 3. The best reduction rate recorded for 80% traffic coverage was 20.14.

Feature entropy which is the amount of information contained in the input 5-tuple and *feature significance* influencing the model analyses were performed and their results are shown in Figures 5.21 and 5.22. Analysis of feature significance was examined for all input data variations (*bits*, *octets*) and all dataset balancing ratios (10%, 20%, 100%). In the context of this research, entropy expresses the average amount of information contained in a feature. The higher the entropy, the greater the randomness, and the lower the entropy, the lower the variation of the values. Entropy is expressed in bits. The results can be interpreted as follows: How many bits are needed to encode the information in an analyzed feature on average? As shown in Figures 5.21 and 5.22 transport protocol and the source port have the least random fields (the most predictable). This could be expected since the transport protocol field contains mostly one of the two values: 6 standing for the TCP protocol, and 17 standing for the UDP protocol. On the other hand, the most random fields are source and destination addresses, and destination port. As expected, only a couple of initial features were truly important for the model to predict the flow size. Even across different balancing ratios, the features were not equally significant. For the *octets* input data and 10% ratio both source and destination IP addresses were the most significant for the TabNet model. For the 20% ratio source and destination ports were the most significant while for the 100% ratio transport protocol was the most significant feature. On the other hand, for the *bits* input data, destination port and transport protocol are important across all balancing ratios while addresses are not relevant at all, and the significance of the source port rises at higher ratios.

Header feature bit number	Header field	Octets [bits]	Bit vector [bits]	Octets 10% ratio	Octets 20% ratio	Octets 100% ratio	Bit vector 10% ratio	Bit vector 20% ratio	Bit vector 100% ratio
0	Source IP address	6.57	0.99	0.02	0.00	0.00	0.00	0.00	0.00
1		6.57	0.99	0.02	0.00	0.00	0.00	0.00	0.00
2		6.57	0.99	0.02	0.00	0.00	0.00	0.00	0.00
3		6.57	0.96	0.02	0.00	0.00	0.00	0.00	0.00
4		6.57	0.97	0.02	0.00	0.00	0.00	0.00	0.00
5		6.57	0.99	0.02	0.00	0.00	0.00	0.00	0.00
6		6.57	0.98	0.02	0.00	0.00	0.00	0.00	0.00
7		6.57	0.98	0.02	0.00	0.00	0.00	0.01	0.00
8		6.05	0.96	0.15	0.01	0.04	0.00	0.01	0.00
9		6.05	0.89	0.15	0.01	0.04	0.00	0.00	0.00
10		6.05	0.76	0.15	0.01	0.04	0.00	0.08	0.00
11		6.05	0.89	0.15	0.01	0.04	0.00	0.00	0.00
12		6.05	0.93	0.15	0.01	0.04	0.00	0.00	0.00
13		6.05	0.90	0.15	0.01	0.04	0.00	0.00	0.00
14		6.05	0.92	0.15	0.01	0.04	0.01	0.00	0.00
15		6.05	0.97	0.15	0.01	0.04	0.01	0.00	0.00
16		7.20	0.88	0.02	0.00	0.00	0.00	0.00	0.00
17		7.20	0.90	0.02	0.00	0.00	0.01	0.00	0.00
18		7.20	0.93	0.02	0.00	0.00	0.00	0.00	0.00
19		7.20	0.99	0.02	0.00	0.00	0.01	0.00	0.00
20		7.20	1.00	0.02	0.00	0.00	0.00	0.01	0.00
21		7.20	1.00	0.02	0.00	0.00	0.00	0.00	0.00
22		7.20	1.00	0.02	0.00	0.00	0.00	0.00	0.00
23		7.20	1.00	0.02	0.00	0.00	0.00	0.00	0.00
24		5.85	0.79	0.01	0.11	0.04	0.00	0.05	0.00
25		5.85	0.84	0.01	0.11	0.04	0.00	0.00	0.00
26		5.85	0.80	0.01	0.11	0.04	0.00	0.00	0.00
27		5.85	0.95	0.01	0.11	0.04	0.02	0.00	0.00
28		5.85	0.94	0.01	0.11	0.04	0.00	0.00	0.00
29		5.85	0.90	0.01	0.11	0.04	0.01	0.00	0.00
30		5.85	0.94	0.01	0.11	0.04	0.00	0.00	0.00
31		5.85	0.93	0.01	0.11	0.04	0.00	0.00	0.00
32	Destination IP address	4.71	0.80	0.03	0.00	0.01	0.00	0.00	0.00
33		4.71	0.83	0.03	0.00	0.01	0.00	0.01	0.00
34		4.71	0.83	0.03	0.00	0.01	0.00	0.00	0.00
35		4.71	0.84	0.03	0.00	0.01	0.01	0.00	0.00
36		4.71	0.80	0.03	0.00	0.01	0.00	0.00	0.00
37		4.71	0.83	0.03	0.00	0.01	0.00	0.01	0.00
38		4.71	0.77	0.03	0.00	0.01	0.00	0.01	0.00
39		4.71	0.79	0.03	0.00	0.01	0.00	0.00	0.00
40		4.24	0.83	0.19	0.00	0.00	0.00	0.00	0.00
41		4.24	0.78	0.19	0.00	0.00	0.00	0.00	0.00
42		4.24	0.79	0.19	0.00	0.00	0.00	0.00	0.00
43		4.24	0.81	0.19	0.00	0.00	0.00	0.00	0.00
44		4.24	0.79	0.19	0.00	0.00	0.00	0.00	0.00
45		4.24	0.80	0.19	0.00	0.00	0.05	0.00	0.00
46		4.24	0.86	0.19	0.00	0.00	0.00	0.01	0.00
47		4.24	0.84	0.19	0.00	0.00	0.00	0.02	0.00
48		7.30	0.90	0.19	0.00	0.01	0.00	0.01	0.00
49		7.30	0.91	0.19	0.00	0.01	0.00	0.00	0.00
50		7.30	0.92	0.19	0.00	0.01	0.00	0.00	0.00
51		7.30	0.99	0.19	0.00	0.01	0.01	0.00	0.00
52		7.30	1.00	0.19	0.00	0.01	0.00	0.00	0.00
53		7.30	1.00	0.19	0.00	0.01	0.00	0.00	0.00
54		7.30	1.00	0.19	0.00	0.01	0.00	0.01	0.00
55		7.30	0.99	0.19	0.00	0.01	0.00	0.00	0.00
56		5.95	0.80	0.02	0.07	0.02	0.00	0.00	0.00
57		5.95	0.84	0.02	0.07	0.02	0.00	0.01	0.00
58		5.95	0.82	0.02	0.07	0.02	0.00	0.01	0.00
59		5.95	0.93	0.02	0.07	0.02	0.01	0.00	0.00
60		5.95	0.93	0.02	0.07	0.02	0.00	0.00	0.00
61		5.95	0.91	0.02	0.07	0.02	0.01	0.00	0.00
62		5.95	0.92	0.02	0.07	0.02	0.00	0.01	0.00
63		5.95	0.93	0.02	0.07	0.02	0.00	0.00	0.00

Figure 5.21: Feature entropy (calculated) and importance (from TabNet) (bits 0-63).

Header feature bit number	Header field	Octets [bits]	Bit vector [bits]	Octets 10% ratio	Octets 20% ratio	Octets 100% ratio	Bit vector 10% ratio	Bit vector 20% ratio	Bit vector 100% ratio
64	Source port	4.76	0.80	0.10	0.11	0.18	0.00	0.00	0.00
65		4.76	0.83	0.10	0.11	0.18	0.00	0.01	0.44
66		4.76	0.83	0.10	0.11	0.18	0.00	0.08	0.01
67		4.76	0.84	0.10	0.11	0.18	0.00	0.00	0.00
68		4.76	0.81	0.10	0.11	0.18	0.00	0.00	0.00
69		4.76	0.84	0.10	0.11	0.18	0.00	0.00	0.00
70		4.76	0.78	0.10	0.11	0.18	0.00	0.00	0.00
71		4.76	0.78	0.10	0.11	0.18	0.00	0.01	0.00
72		4.32	0.83	0.15	0.38	0.01	0.00	0.00	0.00
73		4.32	0.78	0.15	0.38	0.01	0.00	0.00	0.00
74		4.32	0.79	0.15	0.38	0.01	0.00	0.00	0.00
75		4.32	0.82	0.15	0.38	0.01	0.01	0.00	0.09
76		4.32	0.78	0.15	0.38	0.01	0.00	0.00	0.00
77		4.32	0.80	0.15	0.38	0.01	0.00	0.00	0.00
78		4.32	0.86	0.15	0.38	0.01	0.00	0.01	0.00
79		4.32	0.84	0.15	0.38	0.01	0.00	0.00	0.00
80	Destination port	6.42	0.99	0.08	0.29	0.02	0.00	0.00	0.00
81		6.42	0.98	0.08	0.29	0.02	0.00	0.00	0.00
82		6.42	0.98	0.08	0.29	0.02	0.00	0.00	0.00
83		6.42	0.93	0.08	0.29	0.02	0.00	0.00	0.00
84		6.42	0.98	0.08	0.29	0.02	0.00	0.00	0.00
85		6.42	0.99	0.08	0.29	0.02	0.03	0.00	0.07
86		6.42	0.97	0.08	0.29	0.02	0.00	0.00	0.00
87		6.42	0.97	0.08	0.29	0.02	0.00	0.01	0.00
88		5.76	0.94	0.01	0.01	0.00	0.21	0.00	0.00
89		5.76	0.89	0.01	0.01	0.00	0.00	0.19	0.00
90		5.76	0.74	0.01	0.01	0.00	0.01	0.00	0.00
91		5.76	0.88	0.01	0.01	0.00	0.00	0.00	0.00
92		5.76	0.90	0.01	0.01	0.00	0.01	0.09	0.04
93		5.76	0.87	0.01	0.01	0.00	0.00	0.01	0.00
94		5.76	0.89	0.01	0.01	0.00	0.02	0.00	0.00
95		5.76	0.97	0.01	0.01	0.00	0.12	0.00	0.00
96	Transport protocol	1.11	0.00	0.04	0.01	0.67	0.00	0.08	0.00
97		1.11	0.00	0.04	0.01	0.67	0.15	0.00	0.00
98		1.11	0.00	0.04	0.01	0.67	0.05	0.08	0.02
99		1.11	0.99	0.04	0.01	0.67	0.07	0.00	0.16
100		1.11	0.00	0.04	0.01	0.67	0.02	0.09	0.04
101		1.11	1.00	0.04	0.01	0.67	0.09	0.00	0.12
102		1.11	1.00	0.04	0.01	0.67	0.05	0.00	0.00
103		1.11	1.00	0.04	0.01	0.67	0.00	0.01	0.00

Figure 5.22: Feature entropy (calculated) and importance (from TabNet) (bits 64-103).

5.3 Traditional classifiers

This section thoroughly examines classifiers provided by the *scikit-learn* library [87]. These models were applied to solve the classification problem, specifically binary classification, where the output is a binary decision (0 or 1). The decisions indicate whether a flow is classified as an elephant (1) or a mouse (0). All models in this section were tested using three distinct input data representations: *bits*, *octets*, and *raw*.

To perform binary classification, where the model's output determines whether a flow is a mouse or an elephant, it is crucial to label the training dataset appropriately. To achieve this, the elephant flow size threshold was adjusted before the training phase. The step-by-step process for selecting the threshold is described below.

1. **Define Elephant Flow Thresholds**: Identify 25 different elephant flow thresholds based on the largest flows in the training set.
2. **Determine Thresholds by Traffic Coverage**: Instead of setting fixed thresholds, calculate them dynamically by selecting a percentage of the largest flows to achieve specific traffic coverage levels across the network.
3. **Label Training Data**: Assign a label of 1 (elephant) to flows exceeding the selected threshold and 0 (mouse) to all other flows.
4. **Compute Coverage Values**: Use Equation 5.15 to determine the traffic coverage corresponding to each threshold.
5. **Ensure Evenly Spaced Coverage**: The equation was designed to include a training coverage value of exactly 80% while maintaining evenly spaced traffic coverage values.
6. **Apply Coverage Values in Training**: Train the model using the calculated coverage values, adjusting the decision-making process accordingly to optimize classification performance.

$$coverage = 1 - \frac{1}{1.37972966146121546^{i}} \quad \text{for } i \in \{1, \ldots, 25\} \tag{5.15}$$

The procedure of the flow size threshold selection provided the following traffic coverage values:

$$0.275220, 0.474694, 0.619269, 0.724054, \mathbf{0.800000},$$
$$0.855044, 0.894939, 0.923854, 0.944811, 0.960000,$$
$$0.971009, 0.978988, 0.984771, 0.988962, 0.992000,$$
$$0.994202, 0.995798, 0.996954, 0.997792, 0.998400,$$
$$0.998840, 0.999160, 0.999391, 0.999558, 0.999680$$

A key drawback of framing elephant flow prediction as a classification problem is the need to repeat the training and validation process for each predefined traffic coverage value to generate a curve illustrating flow operations reduction.

Additionally, the traffic coverage achieved on the validation dataset is always lower than the coverage used during training. This is because misclassified elephant flows (false negatives) are not accounted for in the final coverage calculation. As a result, models must be trained with higher traffic coverage values than those intended for actual deployment. Therefore, in this section, the traffic coverage metric is referred to as *Resulting Average Traffic Coverage*.

For the classification problem, dataset balancing was not performed explicitly. Instead, sample weights were applied to ensure that both classes contributed equally to the training process. The total sum of sample weights for each class was normalized to the same value, effectively counteracting the natural class imbalance. As a result, the model did not favor the majority class simply due to its higher occurrence in the dataset. All models in this section, except for **KNeighborsClassifier** (which does not support training with explicit sample weights), were trained using sample weights as defined in Equation 5.16.

$$sample\_weight = \sqrt{flow\_size} \tag{5.16}$$

5.3.1 Training

5-fold cross-validation by dividing the dataset into 5 consecutive subsets was conducted. Each subset was used once as a validation set, while the remaining 4 subsets were combined to create the training dataset. Thus, the training dataset consists of 5,213,988 flows, and each validation set contains 1,303,496 flows (80/20 training to evaluation dataset ratio). As a result, each algorithm is trained 5 times for each training traffic coverage. Its performance is then evaluated on the respective validation set. Finally, the average performance of all algorithms across the 5 folds was calculated together with the standard deviation.

Regarding the hyperparameters - all the models in this Section were trained using their default hyperparameters as defined in *scikit-learn* version *1.4.2*. The flowchart of the training phase is presented in Figure 5.23.

5.3.2 Models

Multiple classifier models provided by the `scikit-learn` library [87] were analyzed. The analysis includes the k-nearest neighbourhood (k-NN) classifier, three boosting-based classifiers, and three tree-based classifiers. Additionally, very recent studies demonstrated the possibility of performing real-time inference of tree-based models in networking without requiring additional ML computation units [67], [20], [119], [80], [118], [106]. Therefore, tree-based models were also explored with limited resources (such as maximum depth and the number of trees for forest-based models) to keep the computational expenses under control. Other classifier models from the *scikit-learn* were also analyzed, however, their performance was not satisfactory, therefore they are not included in this Section. A very brief description of each examined classifier is provided below.

1. **KNeighborsClassifier**: k-NN [89] is an instance-based learning algorithm that can be used for both classification and regression tasks. It identifies the k closest samples in the training dataset to a given input sample, using a distance metric such as Euclidean distance. The algorithm then assigns the

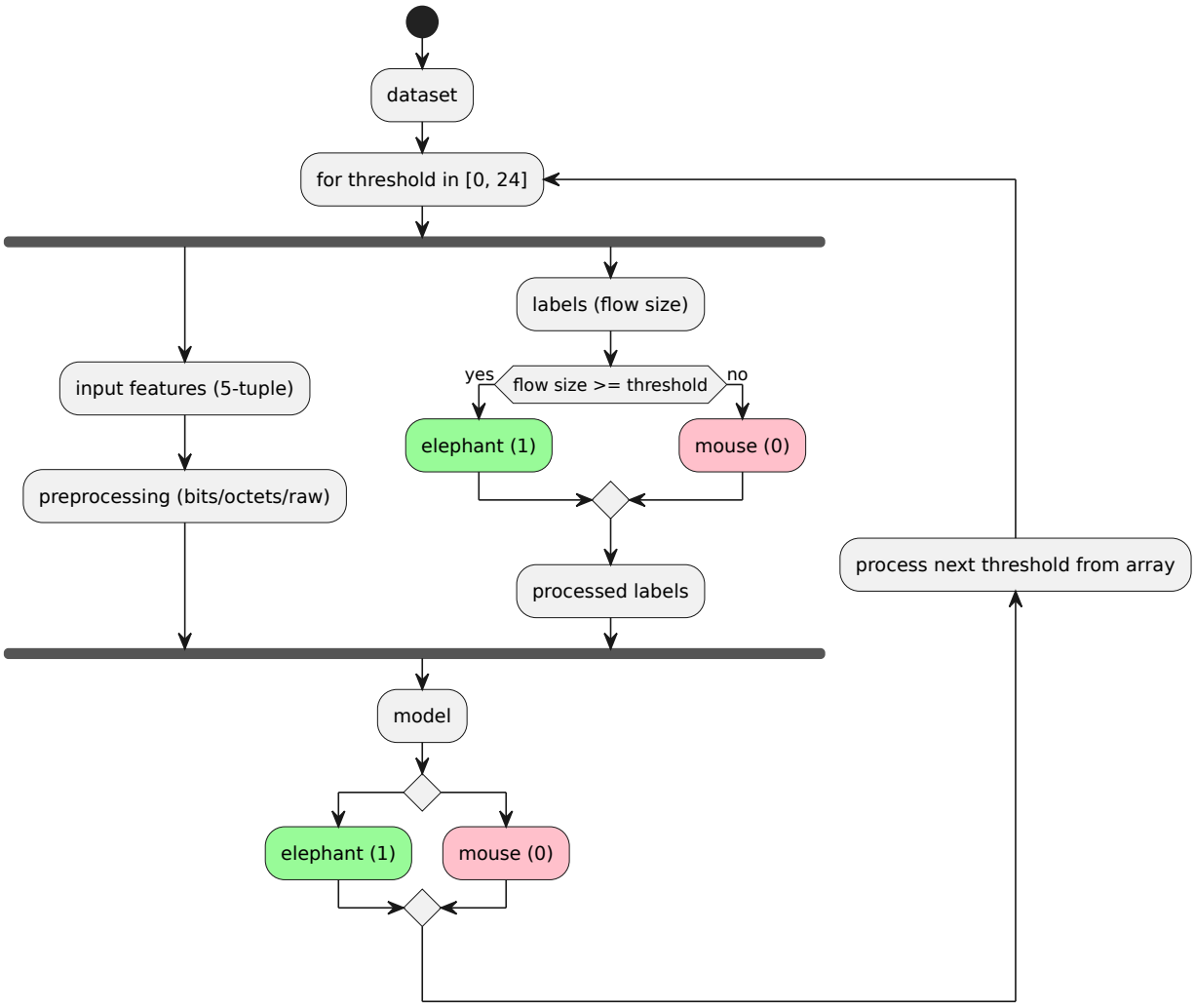

Figure 5.23: Training of the classifications models.

most common class among these neighbors as the prediction. k-NN is a non-parametric and lazy algorithm, meaning it does not have a training phase and relies on the entire training dataset for making predictions. As a result, it can be computationally intensive for large datasets since it requires calculating the distance to all training samples for each new prediction.

2. **AdaBoostClassifier**: AdaBoost, or Adaptive Boosting [31], is an ensemble learning technique that combines the outputs of multiple weak learners to form a strong classifier. It works by sequentially training a series of weak learners, typically decision stumps (simple decision trees with one split), on the training data. After each learner is trained, the dataset is reweighted to give more emphasis to the samples that were incorrectly classified, ensuring subsequent learners focus on harder-to-classify instances. The final model makes predictions based on a weighted sum of all the weak learners' outputs, where each learner's weight corresponds to its accuracy, thus forming a more accurate overall classifier.

3. **GradientBoostingClassifier**: Gradient Boosting [33] is an ensemble learning method that builds models in a sequential manner, where each new model aims to correct the errors made by the pre-

vious models as detailed in Section 4.4. It combines the outputs of multiple weak learners, typically shallow decision trees, by training each tree to fit the residual errors of the combined predictions from the previously trained trees. The process is optimized using gradient descent to minimize a specified loss function, allowing the model to learn more accurately from the data. Gradient Boosting is versatile and can be applied to various tasks due to its ability to handle different types of loss functions.

4. **HistGradientBoostingClassifier**: An efficient variant of Gradient Boosting that uses histogram-based binning to accelerate the training process [59]. By discretizing continuous features into bins and building histograms for each feature, it reduces the computational complexity of identifying optimal splits, making it particularly beneficial for large datasets. This approach decreases memory usage and enhances training speed without compromising performance. HistGradientBoosting supports various loss functions and can natively handle missing values, making it a versatile choice for different types of data.

5. **DecisionTreeClassifier**: Decision Tree is a ML algorithm that works by splitting the dataset into subsets based on the value of the input features as detailed in Section 4.4.

6. **RandomForestClassifier**: Random Forest is an ensemble learning technique that aggregates the predictions of multiple decision trees to enhance overall accuracy and reduce the risk of overfitting as detailed in the Section 4.4.

7. **ExtraTreesClassifier**: Extremely Randomized Trees [36] is an ensemble learning method similar to the RandomForestClassifier. It constructs multiple decision trees using the entire dataset but adds extra randomness by selecting random thresholds for each feature at each split. This added randomness helps to lower the variance of the model, making it less sensitive to the specifics of the training data. Like the RandomForestClassifier, the final prediction is obtained by averaging the predictions of all trees in the ensemble.

8. **DecisionTreeClassifier$X$d**: Decision Tree with maximum depth (*max_depth*) limited to X.

9. **RandomForestClassifier23tXd**: Random Forest consisting of 23 trees (*n_estimators*), with maximum depth (*max_depth*) of each limited to X.

10. **ExtraTreesClassifier23tXd**: Extremely Randomized Trees ensemble consisting of 23 trees (*n_estimators*), with maximum depth (*max_depth*) of each limited to X.

The limitation of both RandomForestClassifier23tXd and ExtraTreesClassifier23tXd was set to 23 trees since for exactly this number, there is no need for packet recirculation as detailed in [106].

Additionally, for all the classification models in Section 5.3 more metrics were calculated. The purpose of presenting those metrics is to show how the typical ML metrics correlate with the ones proposed in Section 5 such as *Flow Operations Reduction* and *Traffic Coverage*. Those metrics are summarized in Table 5.9. Additionally, a brief analysis of when those metrics may be useful is also presented in the form of Table 5.10.

Table 5.9: Traditional ML metrics

Metric name	Description	Formula
TP	Number of true positives.	TP = Number of correctly predicted positive cases
TP	Number of true positives.	TP = Number of correctly predicted positive cases
TN	Number of true negatives.	TN = Number of correctly predicted negative cases
FP	Number of false positives.	FP = Number of negative cases incorrectly predicted as positive
FN	Number of false negatives.	FN = Number of positive cases incorrectly predicted as negative
Accuracy	Accuracy measures the proportion of correctly classified instances among the total number of instances.	$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$ (5.17)
TPR / Recall	True Positive Rate (TPR), also known as Recall or Sensitivity, measures the proportion of actual positive instances that are correctly identified.	$\text{TPR} = \frac{TP}{TP+FN}$ (5.18)
TNR / Specificity	True Negative Rate (TNR), or Specificity, measures the proportion of actual negative instances that are correctly identified.	$\text{TNR} = \frac{TN}{TN+FP}$ (5.19)
FPR	False Positive Rate (FPR) measures the proportion of actual negative instances that are incorrectly classified as positive.	$\text{FPR} = \frac{FP}{FP+TN}$ (5.20)
FNR	False Negative Rate (FNR) measures the proportion of actual positive instances that are incorrectly classified as negative.	$\text{FNR} = \frac{FN}{FN+TP}$ (5.21)
PPV / Precision	Positive Predictive Value (PPV), or Precision, measures the proportion of true positive instances among all instances that are classified as positive.	$\text{PPV} = \frac{TP}{TP+FP}$ (5.22)
NPV	Negative Predictive Value (NPV) measures the proportion of true negative instances among all instances that are classified as negative.	$\text{NPV} = \frac{TN}{TN+FN}$ (5.23)
FDR	False Discovery Rate (FDR) measures the proportion of positive predictions that are false positives.	$\text{FDR} = \frac{FP}{FP+TP}$ (5.24)
FOR	False Omission Rate (FOR) measures the proportion of negative predictions that are actually false negatives.	$\text{FOR} = \frac{FN}{FN+TN}$ (5.25)
FScore	FScore, or F1 Score, is the harmonic mean of Precision and Recall.	$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$ (5.26)
Informedness	Informedness, also known as Bookmaker Informedness (BM), measures the probability that the classifier will make an informed decision as opposed to random guessing.	$\text{BM} = \text{TPR} + \text{TNR} - 1$ (5.27)
MK	Markedness (MK) measures the difference between the TPR and the FDR.	$\text{MK} = \text{PPV} + \text{NPV} - 1$ (5.28)
MCC	Matthews Correlation Coefficient (MCC) is a correlation coefficient between the observed and predicted classifications.	$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$ (5.29)
TS	Threat Score (TS) measures the proportion of correct positive predictions relative to the total number of instances that should have been predicted positive.	$\text{TS} = \frac{TP}{TP+FN+FP}$ (5.30)

Table 5.10: Traditional metrics use cases

Metric name	Use case
Accuracy	Assess the overall effectiveness of a classifier. High accuracy suggests that the model is effectively handling both positive and negative classes. However, accuracy may be deceptive in situations with imbalanced class distributions, as it does not consider the difference in class frequencies.
TPR / Recall	It is essential for assessing a model's capability to identify positive instances, particularly in situations where missing positive cases is costly. A high recall means that the classifier effectively detects most of the positive cases, but it does not take false positives into account.
TNR / Specificity	It is valuable for evaluating a model's ability to identify negative instances, particularly when the cost of false positives is significant. High specificity shows that the classifier is effective at correctly identifying true negatives, but it does not account for false negatives.
FPR	It is important for measuring the rate of false alarms generated by the classifier, which is crucial in areas like fraud detection or medical diagnosis. A low FPR) means that the model produces few false positives, but it does not offer insights into the rate of false negatives.
FNR	This metric measures how frequently the model misses positive instances, which is especially important in situations where false negatives have serious implications. A low FNR indicates that the model effectively captures most positive instances.
PPV / Precision	It is crucial for assessing the accuracy of the model's positive predictions. A high PPV means that the model's positive predictions are highly reliable, which is particularly important in situations where false positives are costly or problematic. However, PPV does not indicate how well the model detects all positive instances.
NPV	It assesses the accuracy of the model's negative predictions, which is vital in situations where correctly identifying negatives is important. A high NPV suggests that the model's negative predictions are reliable, but it does not indicate how well the model detects positive instances.
FDR	It is useful for evaluating the rate at which the model makes incorrect positive predictions, serving as a counterbalance to Precision. A low FDR means that most positive predictions are accurate, which is vital in areas where false positives are costly or problematic. This metric helps researchers and practitioners gauge the reliability of positive results and supports informed decision-making across various fields.
FOR	It helps in assessing how frequently the model incorrectly predicts negatives, which is crucial in situations where false negatives have serious consequences. A low FOR indicates that most negative predictions are accurate, ensuring that the model does not miss many positive cases. FOR is particularly important in scenarios where failing to detect a positive case could lead to significant repercussions. By tracking FOR, researchers can evaluate the thoroughness of their negative predictions and make necessary adjustments to model thresholds or features.
FScore	Particularly useful in cases of imbalanced class distribution, the F1 Score provides a more balanced measure of a classifier's performance by combining Precision and Recall. A high F1 Score indicates that the classifier maintains a good balance between these two metrics, offering a more complete evaluation than either metric alone. This score is commonly used in ML and information retrieval tasks, as it provides a more nuanced assessment of model performance than accuracy, especially in situations where class imbalance could make accuracy misleading.
Informedness	Offers a comprehensive measure of a classifier's performance. A high value indicates that the model is much more effective than random guessing at correctly identifying both positive and negative instances.
MK	It combines PPV and NPV to give a comprehensive measure of the reliability of a classifier's predictions. A high MK indicates that the classifier's predictions are generally accurate and trustworthy.
MCC	It offers a balanced measure of performance, even in the presence of imbalanced class distributions, by providing a single metric that reflects the overall effectiveness of the classifier. A high MCC suggests that the model performs well across all classes, making it a reliable and robust metric for evaluation.
TS	It is valuable for assessing classifier performance in situations where predicting rare events is crucial. A high TS indicates that the model is effective at detecting positive instances while minimizing the number of false positives.

5.3.3 Results

Results for all analyzed metrics across the five validation dataset folds for the k-NN and boosting-based models are shown in Figures 5.24, 5.25, and 5.26. Three elephant thresholds out of the 25 values tested — 80%, 96%, and 99.2% — were selected for presentation. The intensity of the colors in the table fields reflects their value relative to other fields within the same column. As expected, the same training traffic coverage results in completely different traffic coverages (column 1 in Figures 5.24, 5.25, and 5.26). For a proper comparison of the examined models, it was necessary to normalize the metrics against the resulting traffic coverage. Interpolated metric values for three selected resulting traffic coverage levels: 70%, 80%, and 90% are presented in Figure 5.27. After performing interpolation for each fold, the average and standard deviation were calculated. The concept of subsequent figures is the same as in Section 5.1.3 and Section 5.2.2, meaning that the y-axis is on a logarithmic scale, the x-axis covers 50%-100% resulting traffic coverage, the goal is to achieve maximum reduction while maintaining the highest traffic coverage, and **Data** line visualize the ideal performance of a model. The average values of the flow table operations reduction are shown in Figure 5.28. All three input data variations are presented: raw - continuous line, octets - dashed line, bits - dotted line. The results are further divided by input data representation for better visual clarity, and the standard deviation is also included beside the mean. Figure 5.29 present results for the *raw* input data, whereas Figure 5.30 present results for the *octets* input data, and Figure 5.31 present results for the *bits* input data. As it can be seen in Figure 5.31 there is no *KNeighborsClassifier*. These results are missing because the training and validation process for this input data representation and model was not completed within the 5-day limit, as the training was significantly slower compared to other models.

The average traffic coverage on the validation set is consistently lower than the traffic coverage used during training, with the gap varying across different models as can be seen in Figures 5.24, 5.25, and 5.26. For the *KNeighborsClassifier*, this difference is the biggest. In addition to the high *accuracy*, *KNeighborsClassifier* also exhibits a high *FNR*, meaning that traffic from incorrectly classified elephant flows (false negatives) is not accounted for. As stated previously in Section 5 misclassifying an elephant flow significantly affects traffic coverage more than misclassifying mice flows - this is not reflected in the accuracy metric, which treats errors in both classes equally. Therefore, in general, models with higher **FNR** tend to have lower resulting traffic coverage. Figure 5.28 confirms that *KNeighborsClassifier* has the worst performance in terms of *flow operations reduction* for any input data variation. Additionally, to achieve comparable results *KNeighborsClassifier* model required the highest training traffic coverage. In contrast, more efficient models can be trained on lower traffic coverage while still reaching the same resulting coverage. To compare metrics across different models it was crucial to normalize for the same resulting traffic coverage - only then metrics like *accuracy* can be truthfully analyzed - as shown in Figure 5.27. After normalization, *accuracy* is more aligned with the *flow operations reduction*, because normalized *accuracy* compensates for the higher impact of false negatives (elephants misclassified as mouse) over false positives (mouse misclassified as elephants). Very interestingly other metrics such as *TNR* and *NPV* are correlated with a *flow operations reduction*. This is surprising as it appears to contradict the focus on minimizing *FNR*. If minimizing the *FNR* was crucial, it would mean maximizing *TPR*. This is because, in comparisons normalized by traffic coverage, it is *TNR*, not *TPR*, that shows a stronger correlation with performance as the outsized impact of false negatives is

Training traffic coverage: 80%

	Resulting Average Coverage	True Positives	True Negatives	False Positives	False Negatives	TPR (Recall)	TNR (Specificity)	FPR	FNR	PPV (Precision)	NPV	FDR	FOR	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
KNeighborsClassifier, raw, 0	0.277	299	1301903	341	954	0.239	1.000	0.000	0.761	0.467	0.999	0.533	0.001	0.999	0.316	0.238	0.466	0.333	0.188	2036.71
KNeighborsClassifier, raw, 1	0.296	299	1301752	257	1189	0.201	1.000	0.000	0.799	0.538	0.999	0.462	0.001	0.999	0.293	0.201	0.537	0.328	0.171	2344.42
KNeighborsClassifier, raw, 2	0.319	325	1301839	292	1041	0.238	1.000	0.000	0.762	0.527	0.999	0.473	0.001	0.999	0.328	0.238	0.526	0.354	0.196	2112.64
KNeighborsClassifier, raw, 3	0.332	322	1301780	249	1146	0.219	1.000	0.000	0.781	0.564	0.999	0.436	0.001	0.999	0.316	0.219	0.563	0.351	0.188	2282.83
KNeighborsClassifier, raw, 4	0.313	346	1301716	418	1016	0.254	1.000	0.000	0.746	0.453	0.999	0.547	0.001	0.999	0.325	0.254	0.452	0.339	0.194	1706.15
KNeighborsClassifier, octets, 0	0.283	271	1301949	295	982	0.216	1.000	0.000	0.784	0.479	0.999	0.521	0.001	0.999	0.298	0.216	0.478	0.321	0.175	2303.00
KNeighborsClassifier, octets, 1	0.303	299	1301755	254	1189	0.201	1.000	0.000	0.799	0.541	0.999	0.459	0.001	0.999	0.293	0.201	0.540	0.329	0.172	2357.14
KNeighborsClassifier, octets, 2	0.292	291	1301914	217	1075	0.213	1.000	0.000	0.787	0.573	0.999	0.427	0.001	0.999	0.311	0.213	0.572	0.349	0.184	2565.94
KNeighborsClassifier, octets, 3	0.333	291	1301825	204	1177	0.198	1.000	0.000	0.802	0.588	0.999	0.412	0.001	0.999	0.296	0.198	0.587	0.341	0.174	2633.33
KNeighborsClassifier, octets, 4	0.326	334	1301850	284	1028	0.245	1.000	0.000	0.755	0.540	0.999	0.460	0.001	0.999	0.337	0.245	0.540	0.364	0.203	2109.22
AdaBoostClassifier, raw, 0	0.360	359	1295277	6967	894	0.287	0.995	0.005	0.713	0.049	0.999	0.951	0.001	0.994	0.084	0.281	0.048	0.117	0.044	177.93
AdaBoostClassifier, raw, 1	0.386	416	1294603	7406	1072	0.280	0.994	0.006	0.720	0.053	0.999	0.947	0.001	0.993	0.089	0.274	0.052	0.120	0.047	166.64
AdaBoostClassifier, raw, 2	0.436	460	1299087	3044	906	0.337	0.998	0.002	0.663	0.131	0.999	0.869	0.001	0.997	0.189	0.334	0.131	0.209	0.104	372.00
AdaBoostClassifier, raw, 3	0.471	485	1298789	3240	983	0.330	0.998	0.002	0.670	0.130	0.999	0.870	0.001	0.997	0.187	0.328	0.129	0.206	0.103	349.93
AdaBoostClassifier, raw, 4	0.475	547	1294516	7618	815	0.402	0.994	0.006	0.598	0.067	0.999	0.933	0.001	0.994	0.115	0.396	0.066	0.162	0.061	159.64
AdaBoostClassifier, octets, 0	0.474	533	1292988	9256	720	0.425	0.993	0.007	0.575	0.054	0.999	0.946	0.001	0.992	0.097	0.418	0.054	0.150	0.051	133.16
AdaBoostClassifier, octets, 1	0.457	531	1292707	9302	957	0.357	0.993	0.007	0.643	0.054	0.999	0.946	0.001	0.992	0.094	0.350	0.053	0.136	0.049	132.56
AdaBoostClassifier, octets, 2	0.466	511	1293055	9076	855	0.374	0.993	0.007	0.626	0.053	0.999	0.947	0.001	0.992	0.093	0.367	0.053	0.139	0.049	135.97
AdaBoostClassifier, octets, 3	0.492	521	1293175	8854	947	0.355	0.993	0.007	0.645	0.056	0.999	0.944	0.001	0.992	0.096	0.348	0.055	0.138	0.050	139.04
AdaBoostClassifier, octets, 4	0.499	573	1293267	8867	789	0.421	0.993	0.007	0.579	0.061	0.999	0.939	0.001	0.993	0.106	0.414	0.060	0.158	0.056	138.08
AdaBoostClassifier, bits, 0	0.394	433	1287414	14830	820	0.346	0.989	0.011	0.654	0.028	0.999	0.972	0.001	0.988	0.052	0.334	0.028	0.096	0.027	85.40
AdaBoostClassifier, bits, 1	0.407	441	1288156	13853	1047	0.296	0.989	0.011	0.704	0.031	0.999	0.969	0.001	0.989	0.056	0.286	0.030	0.093	0.029	91.19
AdaBoostClassifier, bits, 2	0.413	460	1288861	13270	906	0.337	0.990	0.010	0.663	0.034	0.999	0.966	0.001	0.989	0.061	0.327	0.033	0.103	0.031	94.94
AdaBoostClassifier, bits, 3	0.432	471	1288059	13970	997	0.321	0.989	0.011	0.679	0.033	0.999	0.967	0.001	0.989	0.059	0.310	0.032	0.099	0.031	90.26
AdaBoostClassifier, bits, 4	0.416	454	1290858	11276	908	0.333	0.991	0.009	0.667	0.039	0.999	0.961	0.001	0.991	0.069	0.325	0.038	0.111	0.036	111.12
GradientBoostingClassifier, raw, 0	0.448	493	1298784	3460	760	0.393	0.997	0.003	0.607	0.125	0.999	0.875	0.001	0.997	0.189	0.391	0.124	0.220	0.105	329.75
GradientBoostingClassifier, raw, 1	0.480	558	1299092	2917	930	0.375	0.998	0.002	0.625	0.161	0.999	0.839	0.001	0.997	0.225	0.373	0.160	0.244	0.127	375.11
GradientBoostingClassifier, raw, 2	0.455	497	1299858	2273	869	0.364	0.998	0.002	0.636	0.179	0.999	0.821	0.001	0.998	0.240	0.362	0.179	0.254	0.137	470.58
GradientBoostingClassifier, raw, 3	0.503	545	1300107	1922	923	0.371	0.999	0.001	0.629	0.221	0.999	0.779	0.001	0.998	0.277	0.370	0.220	0.285	0.161	528.37
GradientBoostingClassifier, raw, 4	0.509	612	1299781	2353	750	0.449	0.998	0.002	0.551	0.206	0.999	0.794	0.001	0.998	0.283	0.448	0.206	0.304	0.165	439.63
GradientBoostingClassifier, octets, 0	0.495	570	1296645	5599	683	0.455	0.996	0.004	0.545	0.092	0.999	0.908	0.001	0.995	0.154	0.451	0.092	0.203	0.083	211.30
GradientBoostingClassifier, octets, 1	0.532	658	1296842	5167	830	0.442	0.996	0.004	0.558	0.113	0.999	0.887	0.001	0.995	0.180	0.438	0.112	0.222	0.099	223.78
GradientBoostingClassifier, octets, 2	0.540	629	1296809	5322	737	0.460	0.996	0.004	0.540	0.106	0.999	0.894	0.001	0.995	0.172	0.456	0.105	0.219	0.094	219.04
GradientBoostingClassifier, octets, 3	0.561	648	1296635	5394	820	0.441	0.996	0.004	0.559	0.107	0.999	0.893	0.001	0.995	0.173	0.437	0.107	0.216	0.094	215.74
GradientBoostingClassifier, octets, 4	0.539	667	1297484	4650	695	0.490	0.996	0.004	0.510	0.125	0.999	0.875	0.001	0.996	0.200	0.486	0.125	0.246	0.111	245.16
GradientBoostingClassifier, bits, 0	0.485	560	1293864	8380	693	0.447	0.994	0.006	0.553	0.063	0.999	0.937	0.001	0.993	0.110	0.440	0.062	0.165	0.058	145.81
GradientBoostingClassifier, bits, 1	0.527	658	1294737	7272	830	0.442	0.994	0.006	0.558	0.083	0.999	0.917	0.001	0.994	0.140	0.437	0.082	0.190	0.075	164.38
GradientBoostingClassifier, bits, 2	0.524	610	1293671	8460	756	0.447	0.994	0.006	0.553	0.067	0.999	0.933	0.001	0.993	0.117	0.440	0.067	0.171	0.062	143.72
GradientBoostingClassifier, bits, 3	0.551	635	1294914	7115	833	0.433	0.995	0.005	0.567	0.082	0.999	0.918	0.001	0.994	0.138	0.427	0.081	0.186	0.074	168.19
GradientBoostingClassifier, bits, 4	0.531	669	1294903	7231	693	0.491	0.994	0.006	0.509	0.085	0.999	0.915	0.001	0.994	0.144	0.486	0.084	0.202	0.078	165.00
HistGradientBoostingClassifier, raw, 0	0.540	639	1290799	11445	614	0.510	0.991	0.009	0.490	0.053	1.000	0.947	0.000	0.991	0.096	0.501	0.052	0.162	0.050	107.87
HistGradientBoostingClassifier, raw, 1	0.514	633	1294385	7624	855	0.425	0.994	0.006	0.575	0.077	0.999	0.923	0.001	0.993	0.130	0.420	0.076	0.179	0.069	157.87
HistGradientBoostingClassifier, raw, 2	0.593	717	1291139	10992	649	0.525	0.992	0.008	0.475	0.061	0.999	0.939	0.001	0.991	0.110	0.516	0.061	0.177	0.058	111.32
HistGradientBoostingClassifier, raw, 3	0.541	617	1293647	8382	851	0.420	0.994	0.006	0.580	0.069	0.999	0.931	0.001	0.993	0.118	0.414	0.068	0.168	0.063	144.85
HistGradientBoostingClassifier, raw, 4	0.549	681	1293695	8439	681	0.500	0.994	0.006	0.500	0.075	0.999	0.925	0.001	0.993	0.130	0.494	0.074	0.191	0.069	142.93
HistGradientBoostingClassifier, octets, 0	0.549	687	1296999	5245	566	0.548	0.996	0.004	0.452	0.116	1.000	0.884	0.000	0.996	0.191	0.544	0.115	0.251	0.106	219.74
HistGradientBoostingClassifier, octets, 1	0.579	790	1297256	4753	698	0.531	0.996	0.004	0.469	0.143	0.999	0.857	0.001	0.996	0.225	0.527	0.142	0.274	0.127	235.16
HistGradientBoostingClassifier, octets, 2	0.597	758	1297347	4784	608	0.555	0.996	0.004	0.445	0.137	1.000	0.863	0.000	0.996	0.219	0.551	0.136	0.274	0.123	235.20
HistGradientBoostingClassifier, octets, 3	0.600	759	1297934	4095	709	0.517	0.997	0.003	0.483	0.156	0.999	0.844	0.001	0.996	0.240	0.514	0.156	0.283	0.136	268.54
HistGradientBoostingClassifier, octets, 4	0.579	781	1298388	3746	581	0.573	0.997	0.003	0.427	0.173	1.000	0.827	0.000	0.997	0.265	0.571	0.172	0.313	0.153	287.94
HistGradientBoostingClassifier, bits, 0	0.544	680	1296593	5651	573	0.543	0.996	0.004	0.457	0.107	1.000	0.893	0.000	0.995	0.179	0.538	0.107	0.240	0.098	205.89
HistGradientBoostingClassifier, bits, 1	0.584	775	1297143	4866	713	0.521	0.996	0.004	0.479	0.137	0.999	0.863	0.001	0.996	0.217	0.517	0.137	0.266	0.122	231.08
HistGradientBoostingClassifier, bits, 2	0.587	746	1297219	4912	620	0.546	0.996	0.004	0.454	0.132	1.000	0.868	0.000	0.996	0.212	0.542	0.131	0.267	0.119	230.38
HistGradientBoostingClassifier, bits, 3	0.604	752	1297315	4714	716	0.512	0.996	0.004	0.488	0.138	0.999	0.862	0.001	0.996	0.217	0.509	0.137	0.264	0.122	238.47
HistGradientBoostingClassifier, bits, 4	0.594	784	1297934	4200	578	0.576	0.997	0.003	0.424	0.157	1.000	0.843	0.000	0.996	0.247	0.572	0.157	0.300	0.141	261.54

Figure 5.24: Performance metrics for the 80% training traffic coverage – full results for all 5 folds.

Training traffic coverage: 96%

	Resulting Average Coverage	True Positives	True Negatives	False Positives	False Negatives	TPR (Recall)	TNR (Specificity)	FPR	FNR	PPV (Precision)	NPV	FDR	FOR	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
KNeighborsClassifier, raw, 0	0.438	2616	1287833	3297	9751	0.212	0.997	0.003	0.788	0.442	0.992	0.558	0.008	0.990	0.286	0.209	0.435	0.301	0.167	220.45
KNeighborsClassifier, raw, 1	0.460	2694	1287591	2865	10347	0.207	0.998	0.002	0.793	0.485	0.992	0.515	0.008	0.990	0.290	0.204	0.477	0.312	0.169	234.48
KNeighborsClassifier, raw, 2	0.458	2702	1287967	3201	9627	0.219	0.998	0.002	0.781	0.458	0.993	0.542	0.007	0.990	0.296	0.217	0.450	0.312	0.174	220.82
KNeighborsClassifier, raw, 3	0.467	2830	1286934	2803	10930	0.206	0.998	0.002	0.794	0.502	0.992	0.498	0.008	0.989	0.292	0.203	0.494	0.317	0.171	231.40
KNeighborsClassifier, raw, 4	0.468	2730	1287569	3522	9685	0.220	0.997	0.003	0.780	0.437	0.993	0.563	0.007	0.990	0.292	0.217	0.429	0.305	0.171	208.49
KNeighborsClassifier, octets, 0	0.427	2433	1288340	2790	9934	0.197	0.998	0.002	0.803	0.466	0.992	0.534	0.008	0.990	0.277	0.195	0.458	0.299	0.161	249.57
KNeighborsClassifier, octets, 1	0.464	2579	1287955	2501	10462	0.198	0.998	0.002	0.802	0.508	0.992	0.492	0.008	0.990	0.285	0.196	0.500	0.313	0.166	256.59
KNeighborsClassifier, octets, 2	0.462	2543	1288573	2595	9786	0.206	0.998	0.002	0.794	0.495	0.992	0.505	0.008	0.991	0.291	0.204	0.487	0.316	0.170	253.70
KNeighborsClassifier, octets, 3	0.478	2624	1287492	2245	11136	0.191	0.998	0.002	0.809	0.539	0.991	0.461	0.009	0.990	0.282	0.189	0.530	0.317	0.164	267.71
KNeighborsClassifier, octets, 4	0.466	2633	1288123	2958	9782	0.212	0.998	0.002	0.788	0.471	0.992	0.529	0.008	0.990	0.292	0.210	0.463	0.312	0.171	233.14
AdaBoostClassifier, raw, 0	0.681	5566	1233796	57334	6801	0.450	0.956	0.044	0.550	0.088	0.995	0.912	0.005	0.951	0.148	0.406	0.083	0.184	0.080	20.72
AdaBoostClassifier, raw, 1	0.696	5891	1231456	59000	7150	0.452	0.954	0.046	0.548	0.091	0.994	0.909	0.006	0.949	0.151	0.406	0.085	0.186	0.082	20.09
AdaBoostClassifier, raw, 2	0.729	6754	1207092	84076	5575	0.548	0.935	0.065	0.452	0.074	0.995	0.926	0.005	0.931	0.131	0.483	0.070	0.184	0.070	14.35
AdaBoostClassifier, raw, 3	0.724	6476	1225435	64302	7284	0.471	0.950	0.050	0.529	0.091	0.994	0.909	0.006	0.945	0.153	0.421	0.086	0.190	0.083	18.42
AdaBoostClassifier, raw, 4	0.753	6770	1202298	88783	5645	0.545	0.931	0.069	0.455	0.071	0.995	0.929	0.005	0.928	0.125	0.477	0.066	0.178	0.067	13.64
AdaBoostClassifier, octets, 0	0.690	5444	1226295	64835	6923	0.440	0.950	0.050	0.560	0.077	0.994	0.923	0.006	0.945	0.132	0.390	0.072	0.167	0.071	18.55
AdaBoostClassifier, octets, 1	0.701	5885	1219900	70556	7156	0.451	0.945	0.055	0.549	0.077	0.994	0.923	0.006	0.940	0.132	0.397	0.071	0.168	0.070	17.05
AdaBoostClassifier, octets, 2	0.733	5843	1213810	77358	6486	0.474	0.940	0.060	0.526	0.070	0.995	0.930	0.005	0.936	0.122	0.414	0.065	0.164	0.065	15.67
AdaBoostClassifier, octets, 3	0.726	6161	1221131	68606	7599	0.448	0.947	0.053	0.552	0.082	0.994	0.918	0.006	0.942	0.139	0.395	0.076	0.173	0.075	17.43
AdaBoostClassifier, octets, 4	0.740	5782	1209602	81479	6633	0.466	0.937	0.063	0.534	0.066	0.995	0.934	0.005	0.932	0.116	0.403	0.061	0.156	0.062	14.94
AdaBoostClassifier, bits, 0	0.694	5761	1177259	113871	6606	0.466	0.912	0.088	0.534	0.048	0.994	0.952	0.006	0.908	0.087	0.378	0.043	0.127	0.046	10.90
AdaBoostClassifier, bits, 1	0.712	5830	1190092	100364	7211	0.447	0.922	0.078	0.553	0.055	0.994	0.945	0.006	0.917	0.098	0.369	0.049	0.134	0.051	12.27
AdaBoostClassifier, bits, 2	0.723	5834	1181812	109356	6495	0.473	0.915	0.085	0.527	0.051	0.995	0.949	0.005	0.911	0.092	0.388	0.045	0.132	0.048	11.32
AdaBoostClassifier, bits, 3	0.718	6031	1183521	106216	7729	0.438	0.918	0.082	0.562	0.054	0.994	0.946	0.006	0.913	0.096	0.356	0.047	0.130	0.050	11.61
AdaBoostClassifier, bits, 4	0.741	5732	1182038	109043	6683	0.462	0.916	0.084	0.538	0.050	0.994	0.950	0.006	0.911	0.090	0.377	0.044	0.129	0.047	11.36
GradientBoostingClassifier, raw, 0	0.722	6314	1236842	54288	6053	0.511	0.958	0.042	0.489	0.104	0.995	0.896	0.005	0.954	0.173	0.469	0.099	0.216	0.095	21.51
GradientBoostingClassifier, raw, 1	0.744	6512	1243037	47419	6529	0.499	0.963	0.037	0.501	0.121	0.995	0.879	0.005	0.959	0.194	0.463	0.116	0.231	0.108	24.17
GradientBoostingClassifier, raw, 2	0.745	6620	1238967	52201	5709	0.537	0.960	0.040	0.463	0.113	0.995	0.887	0.005	0.956	0.186	0.497	0.108	0.232	0.103	22.16
GradientBoostingClassifier, raw, 3	0.752	7004	1238064	51673	6756	0.509	0.960	0.040	0.491	0.119	0.995	0.881	0.005	0.955	0.193	0.469	0.114	0.231	0.107	22.21
GradientBoostingClassifier, raw, 4	0.754	6614	1238983	52098	5801	0.533	0.960	0.040	0.467	0.113	0.995	0.887	0.005	0.956	0.186	0.492	0.108	0.231	0.103	22.20
GradientBoostingClassifier, octets, 0	0.700	5910	1245073	46057	6457	0.478	0.964	0.036	0.522	0.114	0.995	0.886	0.005	0.960	0.184	0.442	0.109	0.219	0.101	25.08
GradientBoostingClassifier, octets, 1	0.735	6345	1243190	47266	6696	0.487	0.963	0.037	0.513	0.118	0.995	0.882	0.005	0.959	0.190	0.450	0.113	0.225	0.105	24.31
GradientBoostingClassifier, octets, 2	0.753	6151	1244828	46340	6178	0.499	0.964	0.036	0.501	0.117	0.995	0.883	0.005	0.960	0.190	0.463	0.112	0.228	0.105	24.83
GradientBoostingClassifier, octets, 3	0.750	6537	1243802	45935	7223	0.475	0.964	0.036	0.525	0.125	0.994	0.875	0.006	0.959	0.197	0.439	0.119	0.228	0.110	24.84
GradientBoostingClassifier, octets, 4	0.781	6323	1243585	47496	6092	0.509	0.963	0.037	0.491	0.117	0.995	0.883	0.005	0.959	0.191	0.473	0.113	0.231	0.106	24.22
GradientBoostingClassifier, bits, 0	0.731	6351	1226655	64475	6016	0.514	0.950	0.050	0.486	0.090	0.995	0.910	0.005	0.946	0.153	0.464	0.085	0.198	0.083	18.40
GradientBoostingClassifier, bits, 1	0.764	6726	1228944	61512	6315	0.516	0.952	0.048	0.484	0.099	0.995	0.901	0.005	0.948	0.166	0.468	0.093	0.209	0.090	19.10
GradientBoostingClassifier, bits, 2	0.768	6497	1226193	64975	5832	0.527	0.950	0.050	0.473	0.091	0.995	0.909	0.005	0.946	0.155	0.477	0.086	0.203	0.084	18.24
GradientBoostingClassifier, bits, 3	0.756	6896	1226718	63019	6864	0.501	0.951	0.049	0.499	0.099	0.994	0.901	0.006	0.946	0.165	0.452	0.093	0.205	0.090	18.64
GradientBoostingClassifier, bits, 4	0.795	6718	1223469	67612	5697	0.541	0.948	0.052	0.459	0.090	0.995	0.910	0.005	0.944	0.155	0.489	0.086	0.205	0.084	17.54
HistGradientBoostingClassifier, raw, 0	0.780	6783	1232890	58240	5584	0.548	0.955	0.045	0.452	0.104	0.995	0.896	0.005	0.951	0.175	0.503	0.100	0.224	0.096	20.05
HistGradientBoostingClassifier, raw, 1	0.799	7089	1236477	53979	5952	0.544	0.958	0.042	0.456	0.116	0.995	0.884	0.005	0.954	0.191	0.502	0.111	0.236	0.106	21.35
HistGradientBoostingClassifier, raw, 2	0.823	6973	1235791	55377	5356	0.566	0.957	0.043	0.434	0.112	0.996	0.888	0.004	0.953	0.187	0.523	0.108	0.237	0.103	20.91
HistGradientBoostingClassifier, raw, 3	0.837	7584	1230387	59350	6176	0.551	0.954	0.046	0.449	0.113	0.995	0.887	0.005	0.950	0.188	0.505	0.108	0.234	0.104	19.47
HistGradientBoostingClassifier, raw, 4	0.830	6807	1234243	56838	5608	0.548	0.956	0.044	0.452	0.107	0.995	0.893	0.005	0.952	0.179	0.504	0.102	0.227	0.098	20.48
HistGradientBoostingClassifier, octets, 0	0.780	6908	1251509	39621	5459	0.559	0.969	0.031	0.441	0.148	0.996	0.852	0.004	0.965	0.235	0.528	0.144	0.276	0.133	28.01
HistGradientBoostingClassifier, octets, 1	0.813	7382	1252058	38398	5659	0.566	0.970	0.030	0.434	0.161	0.996	0.839	0.004	0.966	0.251	0.536	0.157	0.290	0.144	28.47
HistGradientBoostingClassifier, octets, 2	0.814	7146	1253290	37878	5183	0.580	0.971	0.029	0.420	0.159	0.996	0.841	0.004	0.967	0.249	0.550	0.155	0.292	0.142	28.95
HistGradientBoostingClassifier, octets, 3	0.831	7754	1252039	37698	6006	0.564	0.971	0.029	0.436	0.171	0.995	0.829	0.005	0.966	0.262	0.534	0.166	0.298	0.151	28.68
HistGradientBoostingClassifier, octets, 4	0.840	7276	1251592	39489	5139	0.586	0.969	0.031	0.414	0.156	0.996	0.844	0.004	0.966	0.246	0.555	0.151	0.290	0.140	27.87
HistGradientBoostingClassifier, bits, 0	0.762	6740	1249481	41649	5627	0.545	0.968	0.032	0.455	0.139	0.996	0.861	0.004	0.964	0.222	0.513	0.135	0.263	0.125	26.94
HistGradientBoostingClassifier, bits, 1	0.808	7077	1250756	39700	5964	0.543	0.969	0.031	0.457	0.151	0.995	0.849	0.005	0.965	0.237	0.512	0.147	0.274	0.134	27.87
HistGradientBoostingClassifier, bits, 2	0.807	6820	1251825	39343	5509	0.553	0.970	0.030	0.447	0.148	0.996	0.852	0.004	0.966	0.233	0.523	0.143	0.274	0.132	28.24
HistGradientBoostingClassifier, bits, 3	0.813	7371	1250602	39135	6389	0.536	0.970	0.030	0.464	0.158	0.995	0.842	0.005	0.965	0.245	0.505	0.153	0.278	0.139	28.03
HistGradientBoostingClassifier, bits, 4	0.827	7042	1250779	40302	5373	0.567	0.969	0.031	0.433	0.149	0.996	0.851	0.004	0.965	0.236	0.536	0.144	0.278	0.134	27.53

Figure 5.25: Performance metrics for the 96% training traffic coverage – full results for all 5 folds.

Training traffic coverage: 99.2%

	Resulting Average Coverage	True Positives	True Negatives	False Positives	False Negatives	TPR (Recall)	TNR (Specificity)	FPR	FNR	PPV (Precision)	NPV	FDR	FOR	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
KNeighborsClassifier, raw, 0	0.630	26293	1191935	28302	56967	0.316	0.977	0.023	0.684	0.482	0.954	0.518	0.046	0.935	0.381	0.293	0.436	0.357	0.236	23.88
KNeighborsClassifier, raw, 1	0.641	27308	1190938	27230	58021	0.320	0.978	0.022	0.680	0.501	0.954	0.499	0.046	0.935	0.390	0.298	0.454	0.368	0.243	23.90
KNeighborsClassifier, raw, 2	0.635	26777	1194302	28081	54337	0.330	0.977	0.023	0.670	0.488	0.956	0.512	0.044	0.937	0.394	0.307	0.445	0.370	0.245	23.76
KNeighborsClassifier, raw, 3	0.629	28234	1186961	26190	62112	0.313	0.978	0.022	0.687	0.519	0.950	0.481	0.050	0.932	0.390	0.291	0.469	0.369	0.242	23.95
KNeighborsClassifier, raw, 4	0.651	27898	1176814	46087	52697	0.346	0.962	0.038	0.654	0.377	0.957	0.623	0.043	0.924	0.361	0.308	0.334	0.321	0.220	17.62
KNeighborsClassifier, octets, 0	0.585	25467	1192599	27638	57793	0.306	0.977	0.023	0.694	0.480	0.954	0.520	0.046	0.934	0.374	0.283	0.433	0.350	0.230	24.55
KNeighborsClassifier, octets, 1	0.639	26097	1191842	26326	59232	0.306	0.978	0.022	0.694	0.498	0.953	0.502	0.047	0.934	0.379	0.284	0.450	0.358	0.234	24.86
KNeighborsClassifier, octets, 2	0.619	25691	1195517	26866	55423	0.317	0.978	0.022	0.683	0.489	0.956	0.511	0.044	0.937	0.384	0.295	0.445	0.362	0.238	24.80
KNeighborsClassifier, octets, 3	0.640	26859	1188001	25150	63487	0.297	0.979	0.021	0.703	0.516	0.949	0.484	0.051	0.932	0.377	0.277	0.466	0.359	0.233	25.06
KNeighborsClassifier, octets, 4	0.644	26441	1182560	40341	54154	0.328	0.967	0.033	0.672	0.396	0.956	0.604	0.044	0.928	0.359	0.295	0.352	0.322	0.219	19.52
AdaBoostClassifier, raw, 0	0.920	55277	961125	259112	27983	0.664	0.788	0.212	0.336	0.176	0.972	0.824	0.028	0.780	0.278	0.452	0.148	0.258	0.161	4.15
AdaBoostClassifier, raw, 1	0.931	57677	961554	256614	27652	0.676	0.789	0.211	0.324	0.184	0.972	0.816	0.028	0.782	0.289	0.465	0.156	0.269	0.169	4.15
AdaBoostClassifier, raw, 2	0.891	55241	971848	250535	25873	0.681	0.795	0.205	0.319	0.181	0.974	0.819	0.026	0.788	0.286	0.476	0.155	0.271	0.167	4.26
AdaBoostClassifier, raw, 3	0.912	59528	957392	255759	30818	0.659	0.789	0.211	0.341	0.189	0.969	0.811	0.031	0.780	0.294	0.448	0.158	0.266	0.172	4.13
AdaBoostClassifier, raw, 4	0.921	56895	933113	289788	23700	0.706	0.763	0.237	0.294	0.164	0.975	0.836	0.025	0.760	0.266	0.469	0.139	0.256	0.154	3.76
AdaBoostClassifier, octets, 0	0.927	58251	932104	288133	25009	0.700	0.764	0.236	0.300	0.168	0.974	0.832	0.026	0.760	0.271	0.463	0.142	0.257	0.157	3.76
AdaBoostClassifier, octets, 1	0.933	60910	930072	288096	24419	0.714	0.764	0.236	0.286	0.175	0.974	0.825	0.026	0.760	0.280	0.477	0.149	0.267	0.163	3.73
AdaBoostClassifier, octets, 2	0.909	59954	917603	304780	21160	0.739	0.751	0.249	0.261	0.164	0.977	0.836	0.023	0.750	0.269	0.490	0.142	0.264	0.155	3.57
AdaBoostClassifier, octets, 3	0.924	62474	942296	270855	27872	0.691	0.777	0.223	0.309	0.187	0.971	0.813	0.029	0.771	0.295	0.468	0.159	0.273	0.173	3.91
AdaBoostClassifier, octets, 4	0.924	57529	928535	294366	23066	0.714	0.759	0.241	0.286	0.163	0.976	0.837	0.024	0.756	0.266	0.473	0.139	0.257	0.153	3.70
AdaBoostClassifier, bits, 0	0.930	62458	867227	353010	20802	0.750	0.711	0.289	0.250	0.150	0.977	0.850	0.023	0.713	0.250	0.461	0.127	0.242	0.143	3.14
AdaBoostClassifier, bits, 1	0.936	63421	879688	338480	21908	0.743	0.722	0.278	0.257	0.158	0.976	0.842	0.024	0.724	0.260	0.465	0.134	0.249	0.150	3.24
AdaBoostClassifier, bits, 2	0.915	60525	889964	332419	20589	0.746	0.728	0.272	0.254	0.154	0.977	0.846	0.023	0.729	0.255	0.474	0.131	0.250	0.146	3.32
AdaBoostClassifier, bits, 3	0.926	67110	878583	334568	23236	0.743	0.724	0.276	0.257	0.167	0.974	0.833	0.026	0.726	0.273	0.467	0.141	0.257	0.158	3.25
AdaBoostClassifier, bits, 4	0.933	60631	877545	345356	19964	0.752	0.718	0.282	0.248	0.149	0.978	0.851	0.022	0.720	0.249	0.470	0.127	0.244	0.142	3.21
GradientBoostingClassifier, raw, 0	0.935	58148	988110	232127	25112	0.698	0.810	0.190	0.302	0.200	0.975	0.800	0.025	0.803	0.311	0.508	0.176	0.299	0.184	4.49
GradientBoostingClassifier, raw, 1	0.924	58400	998635	219533	26929	0.684	0.820	0.180	0.316	0.210	0.974	0.790	0.026	0.811	0.322	0.504	0.184	0.304	0.192	4.69
GradientBoostingClassifier, raw, 2	0.909	55883	1011648	210735	25231	0.689	0.828	0.172	0.311	0.210	0.976	0.790	0.024	0.819	0.321	0.517	0.185	0.309	0.191	4.89
GradientBoostingClassifier, raw, 3	0.926	60955	1005040	208111	29391	0.675	0.828	0.172	0.325	0.227	0.972	0.773	0.028	0.818	0.339	0.503	0.198	0.316	0.204	4.84
GradientBoostingClassifier, raw, 4	0.935	56845	985390	237511	23750	0.705	0.806	0.194	0.295	0.193	0.976	0.807	0.024	0.800	0.303	0.511	0.170	0.294	0.179	4.43
GradientBoostingClassifier, octets, 0	0.918	58197	982529	237708	25063	0.699	0.805	0.195	0.301	0.197	0.975	0.803	0.025	0.798	0.307	0.504	0.172	0.294	0.181	4.41
GradientBoostingClassifier, octets, 1	0.924	59104	992987	225181	26225	0.693	0.815	0.185	0.307	0.208	0.974	0.792	0.026	0.807	0.320	0.508	0.182	0.304	0.190	4.59
GradientBoostingClassifier, octets, 2	0.905	56862	995782	226601	24252	0.701	0.815	0.185	0.299	0.201	0.976	0.799	0.024	0.808	0.312	0.516	0.177	0.302	0.185	4.60
GradientBoostingClassifier, octets, 3	0.928	62027	992392	220759	28319	0.687	0.818	0.182	0.313	0.219	0.972	0.781	0.028	0.809	0.332	0.505	0.192	0.311	0.199	4.61
GradientBoostingClassifier, octets, 4	0.936	57502	983247	239654	23093	0.713	0.804	0.196	0.287	0.194	0.977	0.806	0.023	0.798	0.304	0.517	0.171	0.297	0.180	4.39
GradientBoostingClassifier, bits, 0	0.941	61302	946901	273336	21958	0.736	0.776	0.224	0.264	0.183	0.977	0.817	0.023	0.773	0.293	0.512	0.161	0.287	0.172	3.90
GradientBoostingClassifier, bits, 1	0.936	62388	954496	263672	22941	0.731	0.784	0.216	0.269	0.191	0.977	0.809	0.023	0.780	0.303	0.515	0.168	0.294	0.179	4.00
GradientBoostingClassifier, bits, 2	0.919	60449	953718	268665	20665	0.745	0.780	0.220	0.255	0.184	0.979	0.816	0.021	0.778	0.295	0.525	0.162	0.292	0.173	3.96
GradientBoostingClassifier, bits, 3	0.939	65359	952014	261137	24987	0.723	0.785	0.215	0.277	0.200	0.974	0.800	0.026	0.780	0.314	0.508	0.175	0.298	0.186	3.99
GradientBoostingClassifier, bits, 4	0.946	60601	943013	279888	19994	0.752	0.771	0.229	0.248	0.178	0.979	0.822	0.021	0.770	0.288	0.523	0.157	0.287	0.168	3.83
HistGradientBoostingClassifier, raw, 0	0.937	60661	1002099	218138	22599	0.729	0.821	0.179	0.271	0.218	0.978	0.782	0.022	0.815	0.335	0.550	0.196	0.328	0.201	4.68
HistGradientBoostingClassifier, raw, 1	0.953	62647	997163	221005	22682	0.734	0.819	0.181	0.266	0.221	0.978	0.779	0.022	0.813	0.340	0.553	0.199	0.331	0.205	4.60
HistGradientBoostingClassifier, raw, 2	0.948	60337	1000906	221477	20777	0.744	0.819	0.181	0.256	0.214	0.980	0.786	0.020	0.814	0.333	0.563	0.194	0.330	0.199	4.63
HistGradientBoostingClassifier, raw, 3	0.957	65014	998735	214416	25332	0.720	0.823	0.177	0.280	0.233	0.975	0.767	0.025	0.816	0.352	0.543	0.208	0.336	0.213	4.66
HistGradientBoostingClassifier, raw, 4	0.964	59994	991421	231480	20601	0.744	0.811	0.189	0.256	0.206	0.980	0.794	0.020	0.807	0.322	0.555	0.185	0.321	0.192	4.47
HistGradientBoostingClassifier, octets, 0	0.947	60169	1019876	200361	23091	0.723	0.836	0.164	0.277	0.231	0.978	0.769	0.022	0.829	0.350	0.558	0.209	0.341	0.212	5.00
HistGradientBoostingClassifier, octets, 1	0.953	62003	1022233	195935	23326	0.727	0.839	0.161	0.273	0.240	0.978	0.760	0.022	0.832	0.361	0.566	0.218	0.351	0.220	5.05
HistGradientBoostingClassifier, octets, 2	0.936	59254	1025695	196688	21860	0.731	0.839	0.161	0.269	0.232	0.979	0.768	0.021	0.832	0.352	0.570	0.211	0.346	0.213	5.09
HistGradientBoostingClassifier, octets, 3	0.952	64049	1021756	191395	26297	0.709	0.842	0.158	0.291	0.251	0.975	0.749	0.025	0.833	0.370	0.551	0.226	0.353	0.227	5.10
HistGradientBoostingClassifier, octets, 4	0.956	60105	1008437	214464	20490	0.746	0.825	0.175	0.254	0.219	0.980	0.781	0.020	0.820	0.338	0.570	0.199	0.337	0.204	4.75
HistGradientBoostingClassifier, bits, 0	0.936	62022	999940	220297	21238	0.745	0.819	0.181	0.255	0.220	0.979	0.780	0.021	0.815	0.339	0.564	0.199	0.335	0.204	4.62
HistGradientBoostingClassifier, bits, 1	0.951	63071	1009004	209164	22258	0.739	0.828	0.172	0.261	0.232	0.978	0.768	0.022	0.822	0.353	0.567	0.210	0.345	0.214	4.79
HistGradientBoostingClassifier, bits, 2	0.934	60175	1008641	213742	20939	0.742	0.825	0.175	0.258	0.220	0.980	0.780	0.020	0.820	0.339	0.567	0.199	0.336	0.204	4.76
HistGradientBoostingClassifier, bits, 3	0.951	65916	1005201	207950	24430	0.730	0.829	0.171	0.270	0.241	0.976	0.759	0.024	0.822	0.362	0.558	0.217	0.348	0.221	4.76
HistGradientBoostingClassifier, bits, 4	0.957	60674	1000358	222543	19921	0.753	0.818	0.182	0.247	0.214	0.980	0.786	0.020	0.814	0.334	0.571	0.195	0.333	0.200	4.60

Figure 5.26: Performance metrics for the 99.2% training traffic coverage – full results for all 5 folds.

	Resulting Average Coverage	Training Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
KNeighborsClassifier, raw	0.7000	0.9952	0.43 ± 0.01	0.95 ± 0.01	0.53 ± 0.06	0.93 ± 0.01	0.89 ± 0.01	0.48 ± 0.03	0.39 ± 0.01	0.46 ± 0.05	0.42 ± 0.03	0.31 ± 0.02	11.31 ± 0.96
KNeighborsClassifier, octets	0.7000	0.9953	0.44 ± 0.05	0.95 ± 0.01	0.54 ± 0.05	0.93 ± 0.01	0.89 ± 0.02	0.48 ± 0.04	0.39 ± 0.04	0.47 ± 0.04	0.43 ± 0.03	0.32 ± 0.04	11.16 ± 2.66
AdaBoostClassifier, raw	0.7000	0.9569	0.47 ± 0.02	0.95 ± 0.01	0.09 ± 0.01	0.99 ± 0.00	0.95 ± 0.00	0.14 ± 0.01	0.42 ± 0.02	0.08 ± 0.01	0.18 ± 0.00	0.08 ± 0.00	21.96 ± 2.56
AdaBoostClassifier, octets	0.7000	0.9548	0.43 ± 0.03	0.95 ± 0.01	0.07 ± 0.01	0.99 ± 0.00	0.95 ± 0.01	0.12 ± 0.01	0.38 ± 0.02	0.07 ± 0.01	0.16 ± 0.01	0.07 ± 0.01	20.36 ± 2.78
AdaBoostClassifier, bits	0.7000	0.9558	0.43 ± 0.03	0.92 ± 0.01	0.05 ± 0.00	0.99 ± 0.00	0.92 ± 0.01	0.09 ± 0.00	0.36 ± 0.02	0.04 ± 0.00	0.13 ± 0.00	0.05 ± 0.00	13.21 ± 1.78
GradientBoostingClassifier, raw	0.7000	0.9519	0.44 ± 0.03	0.97 ± 0.00	0.12 ± 0.01	1.00 ± 0.00	0.97 ± 0.00	0.19 ± 0.01	0.42 ± 0.02	0.11 ± 0.01	0.22 ± 0.01	0.10 ± 0.01	36.98 ± 5.52
GradientBoostingClassifier, octets	0.7000	0.9506	0.42 ± 0.04	0.98 ± 0.01	0.13 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.20 ± 0.01	0.39 ± 0.03	0.13 ± 0.01	0.22 ± 0.00	0.11 ± 0.01	44.26 ± 12.40
GradientBoostingClassifier, bits	0.7000	0.9464	0.42 ± 0.03	0.97 ± 0.01	0.09 ± 0.00	1.00 ± 0.00	0.97 ± 0.01	0.15 ± 0.00	0.39 ± 0.03	0.09 ± 0.00	0.18 ± 0.00	0.08 ± 0.00	33.18 ± 6.82
HistGradientBoostingClassifier, raw	0.7000	0.9275	0.46 ± 0.02	0.98 ± 0.00	0.10 ± 0.01	1.00 ± 0.00	0.98 ± 0.01	0.16 ± 0.01	0.43 ± 0.02	0.09 ± 0.01	0.20 ± 0.01	0.09 ± 0.01	45.87 ± 8.37
HistGradientBoostingClassifier, octets	0.7000	0.8984	0.54 ± 0.04	0.99 ± 0.00	0.17 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.26 ± 0.02	0.53 ± 0.04	0.17 ± 0.02	0.30 ± 0.02	0.15 ± 0.01	110.24 ± 30.49
HistGradientBoostingClassifier, bits	0.7000	0.9043	0.52 ± 0.04	0.99 ± 0.00	0.15 ± 0.01	1.00 ± 0.00	0.99 ± 0.00	0.24 ± 0.02	0.51 ± 0.04	0.15 ± 0.01	0.28 ± 0.02	0.13 ± 0.01	92.38 ± 25.44
KNeighborsClassifier, raw	0.8000	0.9982	0.66 ± 0.02	0.89 ± 0.01	0.68 ± 0.05	0.88 ± 0.01	0.83 ± 0.01	0.67 ± 0.02	0.55 ± 0.01	0.56 ± 0.03	0.56 ± 0.02	0.51 ± 0.03	4.05 ± 0.27
KNeighborsClassifier, octets	0.8000	0.9977	0.63 ± 0.03	0.91 ± 0.01	0.65 ± 0.04	0.90 ± 0.01	0.85 ± 0.01	0.64 ± 0.03	0.54 ± 0.02	0.55 ± 0.03	0.54 ± 0.02	0.47 ± 0.03	4.90 ± 0.58
AdaBoostClassifier, raw	0.8000	0.9673	0.63 ± 0.02	0.90 ± 0.01	0.08 ± 0.00	0.99 ± 0.00	0.90 ± 0.01	0.14 ± 0.01	0.53 ± 0.02	0.07 ± 0.00	0.20 ± 0.01	0.07 ± 0.00	10.65 ± 0.68
AdaBoostClassifier, octets	0.8000	0.9712	0.59 ± 0.02	0.90 ± 0.01	0.08 ± 0.01	0.99 ± 0.00	0.89 ± 0.01	0.14 ± 0.01	0.49 ± 0.01	0.07 ± 0.01	0.19 ± 0.01	0.08 ± 0.01	9.54 ± 0.95
AdaBoostClassifier, bits	0.8000	0.9742	0.58 ± 0.02	0.87 ± 0.01	0.07 ± 0.01	0.99 ± 0.00	0.86 ± 0.01	0.13 ± 0.01	0.45 ± 0.02	0.06 ± 0.01	0.17 ± 0.01	0.07 ± 0.01	7.26 ± 0.70
GradientBoostingClassifier, raw	0.8000	0.9675	0.60 ± 0.02	0.93 ± 0.01	0.11 ± 0.01	0.99 ± 0.00	0.93 ± 0.01	0.18 ± 0.01	0.53 ± 0.01	0.10 ± 0.01	0.23 ± 0.00	0.10 ± 0.00	15.03 ± 1.57
GradientBoostingClassifier, octets	0.8000	0.9677	0.57 ± 0.03	0.94 ± 0.01	0.11 ± 0.01	0.99 ± 0.00	0.93 ± 0.01	0.19 ± 0.01	0.51 ± 0.02	0.11 ± 0.00	0.24 ± 0.00	0.10 ± 0.00	16.25 ± 2.67
GradientBoostingClassifier, bits	0.8000	0.9666	0.58 ± 0.03	0.93 ± 0.01	0.09 ± 0.00	0.99 ± 0.00	0.92 ± 0.01	0.16 ± 0.01	0.51 ± 0.02	0.09 ± 0.00	0.21 ± 0.00	0.09 ± 0.00	13.56 ± 2.30
HistGradientBoostingClassifier, raw	0.8000	0.9547	0.54 ± 0.02	0.96 ± 0.01	0.11 ± 0.01	1.00 ± 0.00	0.96 ± 0.01	0.18 ± 0.01	0.50 ± 0.02	0.10 ± 0.00	0.23 ± 0.01	0.10 ± 0.00	23.70 ± 3.82
HistGradientBoostingClassifier, octets	0.8000	0.9547	0.54 ± 0.02	0.97 ± 0.01	0.16 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.25 ± 0.01	0.51 ± 0.02	0.16 ± 0.01	0.28 ± 0.00	0.14 ± 0.00	36.49 ± 7.73
HistGradientBoostingClassifier, bits	0.8000	0.9577	0.54 ± 0.02	0.97 ± 0.01	0.15 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.23 ± 0.01	0.51 ± 0.02	0.15 ± 0.01	0.27 ± 0.00	0.13 ± 0.00	31.47 ± 6.37
KNeighborsClassifier, raw	0.9000	0.9992	0.75 ± 0.01	0.76 ± 0.01	0.74 ± 0.03	0.77 ± 0.03	0.76 ± 0.00	0.75 ± 0.02	0.51 ± 0.00	0.51 ± 0.00	0.51 ± 0.00	0.60 ± 0.02	2.11 ± 0.11
KNeighborsClassifier, octets	0.9000	0.9990	0.73 ± 0.02	0.81 ± 0.02	0.73 ± 0.03	0.82 ± 0.03	0.78 ± 0.01	0.73 ± 0.01	0.55 ± 0.01	0.55 ± 0.01	0.55 ± 0.01	0.58 ± 0.01	2.48 ± 0.19
AdaBoostClassifier, raw	0.9000	0.9892	0.69 ± 0.03	0.81 ± 0.02	0.16 ± 0.03	0.98 ± 0.01	0.80 ± 0.02	0.25 ± 0.04	0.49 ± 0.03	0.14 ± 0.02	0.26 ± 0.02	0.15 ± 0.03	4.70 ± 0.48
AdaBoostClassifier, octets	0.9000	0.9875	0.71 ± 0.01	0.81 ± 0.02	0.13 ± 0.02	0.98 ± 0.00	0.80 ± 0.02	0.22 ± 0.03	0.51 ± 0.01	0.12 ± 0.02	0.25 ± 0.01	0.13 ± 0.02	4.70 ± 0.49
AdaBoostClassifier, bits	0.9000	0.9887	0.69 ± 0.01	0.77 ± 0.01	0.12 ± 0.01	0.98 ± 0.00	0.77 ± 0.01	0.21 ± 0.01	0.46 ± 0.01	0.11 ± 0.01	0.22 ± 0.01	0.12 ± 0.01	4.05 ± 0.24
GradientBoostingClassifier, raw	0.9000	0.9870	0.69 ± 0.01	0.84 ± 0.01	0.15 ± 0.02	0.99 ± 0.00	0.83 ± 0.01	0.24 ± 0.02	0.53 ± 0.01	0.13 ± 0.01	0.26 ± 0.02	0.14 ± 0.02	5.59 ± 0.23
GradientBoostingClassifier, octets	0.9000	0.9858	0.68 ± 0.01	0.85 ± 0.02	0.14 ± 0.02	0.99 ± 0.00	0.85 ± 0.02	0.24 ± 0.03	0.53 ± 0.01	0.13 ± 0.02	0.26 ± 0.01	0.14 ± 0.02	6.19 ± 0.89
GradientBoostingClassifier, bits	0.9000	0.9856	0.67 ± 0.01	0.85 ± 0.02	0.14 ± 0.01	0.99 ± 0.00	0.84 ± 0.02	0.22 ± 0.02	0.51 ± 0.01	0.12 ± 0.01	0.25 ± 0.01	0.13 ± 0.01	5.99 ± 0.63
HistGradientBoostingClassifier, raw	0.9000	0.9796	0.65 ± 0.01	0.90 ± 0.01	0.13 ± 0.01	0.99 ± 0.00	0.90 ± 0.01	0.22 ± 0.01	0.55 ± 0.01	0.12 ± 0.01	0.26 ± 0.01	0.12 ± 0.01	9.28 ± 0.99
HistGradientBoostingClassifier, octets	0.9000	0.9798	0.64 ± 0.01	0.92 ± 0.01	0.16 ± 0.01	0.99 ± 0.00	0.91 ± 0.01	0.25 ± 0.01	0.56 ± 0.01	0.15 ± 0.01	0.29 ± 0.01	0.15 ± 0.01	11.15 ± 1.14
HistGradientBoostingClassifier, bits	0.9000	0.9810	0.65 ± 0.02	0.91 ± 0.01	0.15 ± 0.01	0.99 ± 0.00	0.90 ± 0.01	0.24 ± 0.01	0.55 ± 0.01	0.14 ± 0.01	0.28 ± 0.00	0.14 ± 0.01	9.60 ± 1.53

Figure 5.27: Performance metrics for 70%, 80%, and 90% resulting traffic coverage values across all 5 folds.

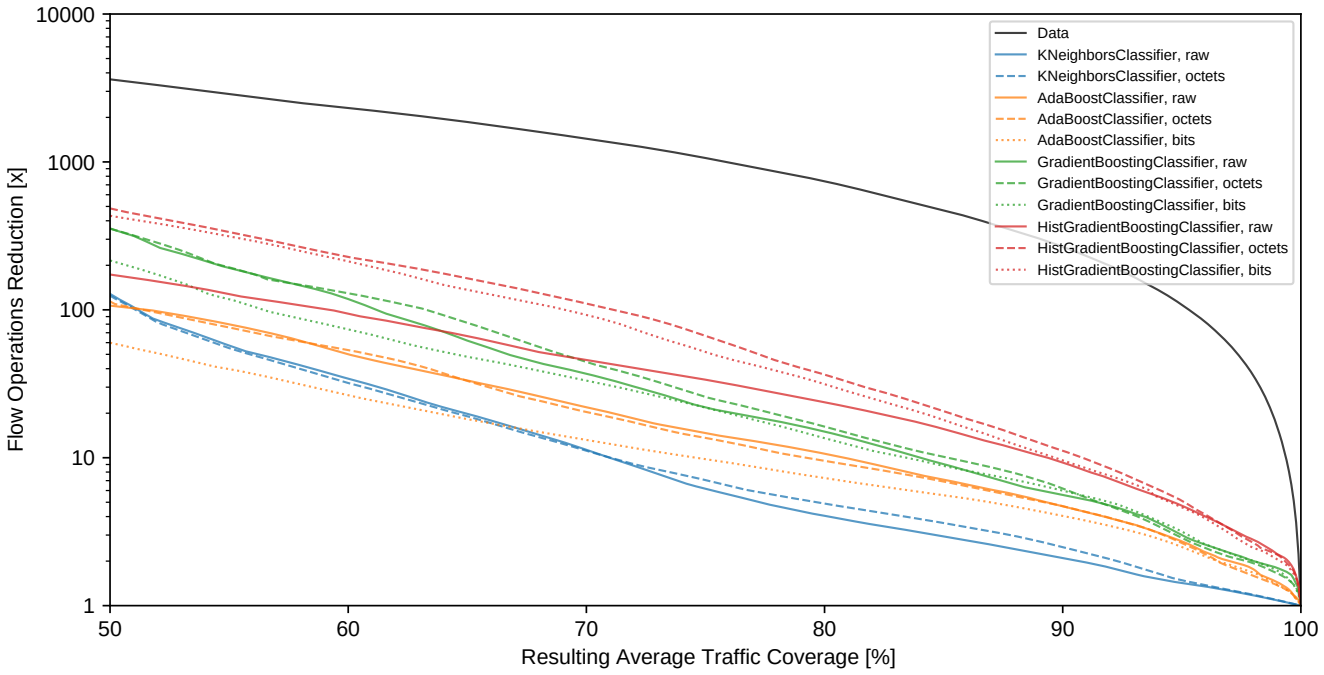

Figure 5.28: Flow operations reduction for the resulting traffic coverage between 50% and 100%.

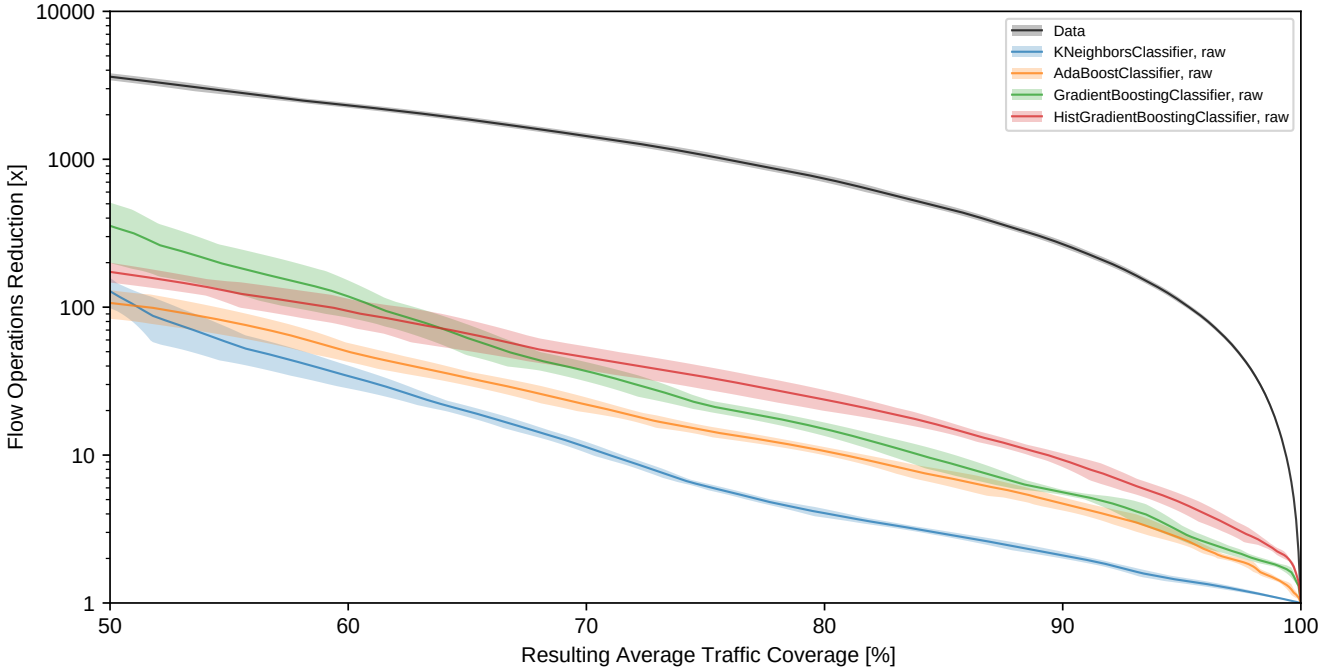

Figure 5.29: Flow operations reduction with standard deviation (shaded line) for *raw* input data.

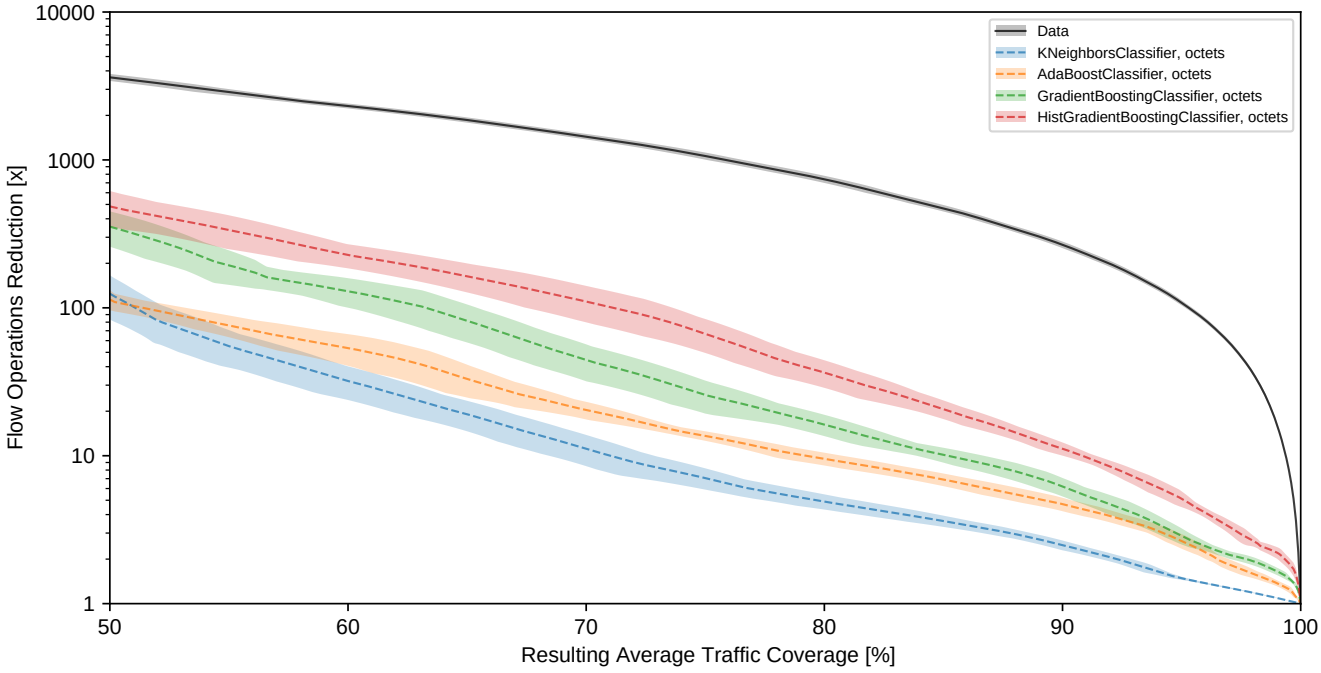

Figure 5.30: Flow operations reduction with standard deviation (shaded line) for *octets* input data.

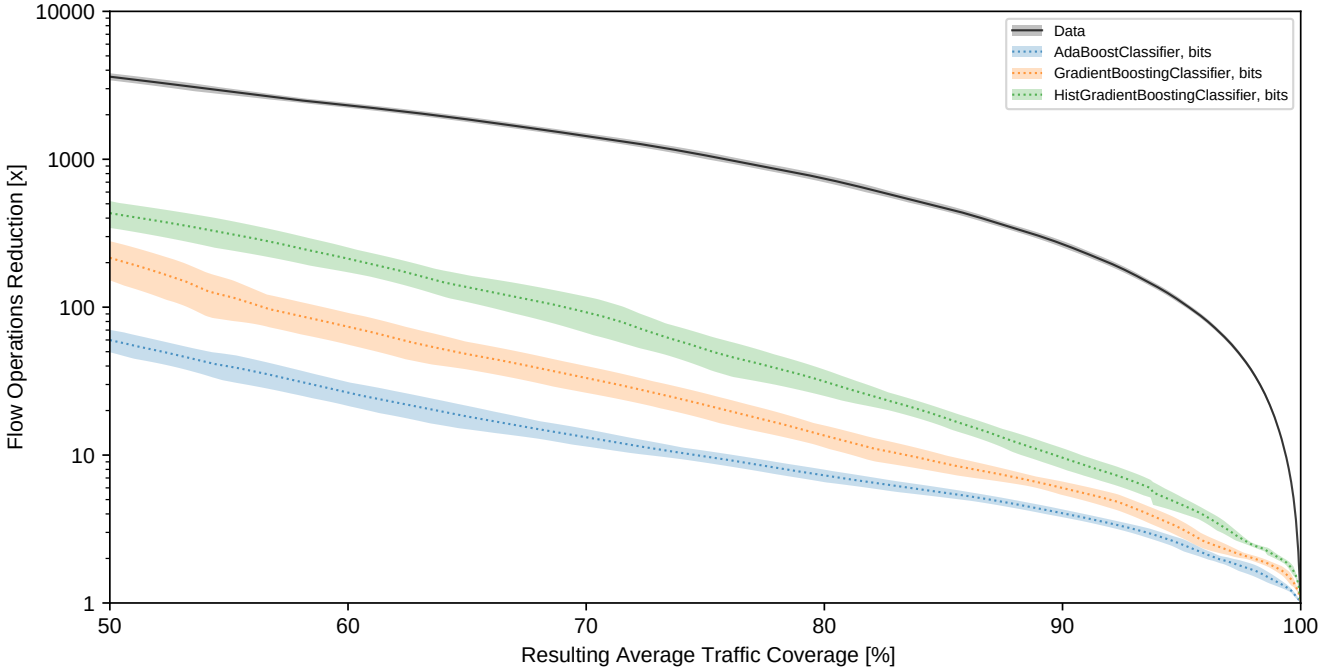

Figure 5.31: Flow operations reduction with standard deviation (shaded line) for *bits* input data.

already mitigated through normalization by resulting traffic coverage. With the desired traffic coverage ensured, the goal shifts to reducing the number of flow operations specifically, minimizing the classification of mice flows as elephants (false positives). Reducing false positives directly translates to maximizing *TNR* and *NPV* as detailed in Equation 5.19 and 5.23. **In summary, traditional classification metrics are insufficient for evaluating elephant flow classification performance. These metrics only become meaningful when compared within the same resulting traffic coverage. Therefore, results must be normalized before accurately comparing *accuracy* or other performance metrics across models.**

As for the performance of the classifiers, as shown in Figure 5.27 the best flow classification results are provided by the *HistGradientBoostingClassifier*. This model is performing the best across the entire range of resulting traffic coverage as shown in 5.28. For 80% of resulting traffic coverage, *HistGradientBoostingClassifier* with *octets* input data representation reduces the number of flow operations by a factor of 36.49 ± 7.73. The worst performance on the other hand is achieved by *KNeighborsClassifier*. This could be expected since as stated at the beginning of this Section, this model was unable to take sample weights into account during the training phase. Apart from the flow classification metrics, *HistGradientBoostingClassifier* also presents very high *accuracy* - 91%, 97%, and 99% for 90%, 80%, and 70% resulting traffic coverage. In contrast, the *KNeighborsClassifier* achieves the accuracy of 84.8% for the *octets* input format which translates to significantly lower flow table metrics. Therefore, elephant flow classifier models need to achieve accuracy higher than 90% to be considered useful in traffic engineering and QoS applications. Not only were performance metrics such as *accuracy* or flow classification metrics the best for the *HistGradientBoostingClassifier* model. *HistGradientBoostingClassifier* had significantly lower training and inference times due to a more efficient implementation of the Gradient Boosting algorithm. On the other hand, *KNeighborsClassifier* was the worst also when it comes to the training and inference time. Specifically for the *bits* input data representation the training did not finish within the 5-day limit. The choice of data representation (*raw*, *octets*, *bits*) plays a crucial role in classifier performance. In general, using octets or bits results in better performance for most classifiers. The octets representation stands out, offering a substantial performance boost while keeping the total number of features low, which helps minimize memory usage and computation time.

Performance metrics of the tree-based classifiers for the normalized resulting traffic coverage 70%, 80%, and 90% are presented in Figures 5.32, 5.35, 5.38, 5.33, 5.36, 5.39, 5.34, 5.37, and 5.40. Exactly like for the non-tree-based models, after interpolation for each fold, mean and standard deviation across all folds were calculated. Figure 5.41 present results for the *raw* input data, whereas Figure 5.42 present results for the *octets* input data, and Figure 5.43 present results for the *bits* input data. Following that, Figure 5.44 illustrates the combined performance of all the analyzed tree-based models at 70%, 80%, and 90% resulting traffic coverages. Additionally, 5.45 shows the training traffic coverage required to reach the desired resulting traffic coverage, along with the corresponding traditional accuracy metric values.

The average number of leaves in trees of analyzed models is summarized in Figure 5.46. This summary shows that models with default, unconstrained settings tend to produce trees with a very large number of leaves. In these cases, nodes are expanded until each leaf is pure or contains fewer samples than the minimum required, leading to fully grown, unpruned trees, which can become extremely large depending

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
DecisionTreeClassifier10d, raw	0.7000	0.9387	0.45 ± 0.02	0.98 ± 0.00	0.13 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.20 ± 0.01	0.43 ± 0.02	0.13 ± 0.01	0.23 ± 0.01	0.11 ± 0.00	51.43 ± 6.33
DecisionTreeClassifier10d, octets	0.7000	0.9260	0.48 ± 0.02	0.98 ± 0.01	0.11 ± 0.00	1.00 ± 0.00	0.98 ± 0.01	0.18 ± 0.00	0.46 ± 0.02	0.11 ± 0.00	0.23 ± 0.00	0.10 ± 0.00	54.14 ± 13.58
DecisionTreeClassifier10d, bits	0.7000	0.8796	0.58 ± 0.02	0.98 ± 0.00	0.06 ± 0.00	1.00 ± 0.00	0.98 ± 0.00	0.10 ± 0.00	0.55 ± 0.02	0.06 ± 0.00	0.18 ± 0.00	0.06 ± 0.00	42.69 ± 2.66
DecisionTreeClassifier15d, raw	0.7000	0.9044	0.54 ± 0.02	0.99 ± 0.00	0.12 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.19 ± 0.01	0.53 ± 0.02	0.11 ± 0.01	0.25 ± 0.01	0.11 ± 0.00	65.85 ± 12.07
DecisionTreeClassifier15d, octets	0.7000	0.9025	0.55 ± 0.02	0.99 ± 0.00	0.13 ± 0.01	1.00 ± 0.00	0.99 ± 0.00	0.22 ± 0.02	0.54 ± 0.02	0.13 ± 0.01	0.27 ± 0.02	0.12 ± 0.01	78.21 ± 14.41
DecisionTreeClassifier15d, bits	0.7000	0.8634	0.62 ± 0.02	0.99 ± 0.00	0.11 ± 0.01	1.00 ± 0.00	0.99 ± 0.00	0.19 ± 0.01	0.61 ± 0.02	0.11 ± 0.01	0.26 ± 0.01	0.10 ± 0.01	90.69 ± 10.21
DecisionTreeClassifier20d, raw	0.7000	0.9341	0.50 ± 0.01	0.98 ± 0.00	0.16 ± 0.02	1.00 ± 0.00	0.98 ± 0.00	0.24 ± 0.03	0.48 ± 0.01	0.15 ± 0.02	0.27 ± 0.02	0.13 ± 0.02	60.84 ± 15.06
DecisionTreeClassifier20d, octets	0.7000	0.9187	0.52 ± 0.01	0.99 ± 0.00	0.18 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.26 ± 0.02	0.51 ± 0.01	0.17 ± 0.02	0.30 ± 0.02	0.15 ± 0.02	86.04 ± 23.27
DecisionTreeClassifier20d, bits	0.7000	0.9049	0.55 ± 0.01	0.99 ± 0.00	0.19 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.28 ± 0.02	0.54 ± 0.01	0.18 ± 0.02	0.32 ± 0.01	0.16 ± 0.01	106.59 ± 26.27
DecisionTreeClassifier25d, raw	0.7000	0.9562	0.50 ± 0.01	0.98 ± 0.00	0.19 ± 0.03	1.00 ± 0.00	0.98 ± 0.00	0.27 ± 0.03	0.48 ± 0.01	0.18 ± 0.03	0.29 ± 0.02	0.16 ± 0.02	43.10 ± 7.01
DecisionTreeClassifier25d, octets	0.7000	0.9398	0.48 ± 0.01	0.99 ± 0.00	0.21 ± 0.03	1.00 ± 0.00	0.98 ± 0.00	0.29 ± 0.03	0.47 ± 0.01	0.20 ± 0.03	0.31 ± 0.03	0.17 ± 0.02	74.63 ± 20.33
DecisionTreeClassifier25d, bits	0.7000	0.9537	0.45 ± 0.01	0.99 ± 0.00	0.24 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.32 ± 0.01	0.44 ± 0.01	0.24 ± 0.01	0.32 ± 0.01	0.19 ± 0.01	66.96 ± 17.72
DecisionTreeClassifier30d, raw	0.7000	0.9745	0.47 ± 0.02	0.97 ± 0.01	0.22 ± 0.03	0.99 ± 0.00	0.96 ± 0.01	0.30 ± 0.03	0.44 ± 0.01	0.21 ± 0.03	0.30 ± 0.02	0.18 ± 0.02	28.05 ± 5.31
DecisionTreeClassifier30d, octets	0.7000	0.9595	0.46 ± 0.02	0.99 ± 0.00	0.25 ± 0.02	0.99 ± 0.00	0.98 ± 0.00	0.32 ± 0.02	0.44 ± 0.02	0.24 ± 0.02	0.33 ± 0.02	0.19 ± 0.01	57.04 ± 14.60
DecisionTreeClassifier30d, bits	0.7000	0.9758	0.41 ± 0.01	0.98 ± 0.00	0.27 ± 0.03	0.99 ± 0.00	0.97 ± 0.01	0.32 ± 0.02	0.39 ± 0.01	0.25 ± 0.03	0.31 ± 0.01	0.19 ± 0.01	35.87 ± 6.75
DecisionTreeClassifier, raw	0.7000	0.9877	0.42 ± 0.02	0.96 ± 0.01	0.30 ± 0.04	0.97 ± 0.00	0.93 ± 0.01	0.35 ± 0.03	0.38 ± 0.02	0.27 ± 0.04	0.32 ± 0.02	0.21 ± 0.02	17.48 ± 2.43
DecisionTreeClassifier, octets	0.7000	0.9735	0.42 ± 0.01	0.98 ± 0.01	0.30 ± 0.02	0.99 ± 0.00	0.97 ± 0.01	0.35 ± 0.01	0.40 ± 0.01	0.28 ± 0.02	0.34 ± 0.01	0.21 ± 0.01	43.91 ± 11.83
DecisionTreeClassifier, bits	0.7000	0.9813	0.42 ± 0.02	0.97 ± 0.01	0.29 ± 0.04	0.98 ± 0.01	0.96 ± 0.01	0.34 ± 0.03	0.39 ± 0.01	0.27 ± 0.03	0.32 ± 0.02	0.20 ± 0.02	28.12 ± 7.54

Figure 5.32: Performance metrics for DecisionTreeClassifier for 70% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
RandomForestClassifier23t10d, raw	0.7000	0.9428	0.44 ± 0.03	0.98 ± 0.00	0.14 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.21 ± 0.01	0.42 ± 0.02	0.14 ± 0.01	0.24 ± 0.01	0.12 ± 0.01	53.99 ± 8.34
RandomForestClassifier23t10d, octets	0.7000	0.9420	0.42 ± 0.03	0.99 ± 0.00	0.23 ± 0.03	1.00 ± 0.00	0.99 ± 0.01	0.29 ± 0.03	0.41 ± 0.03	0.22 ± 0.04	0.30 ± 0.02	0.17 ± 0.02	92.33 ± 28.84
RandomForestClassifier23t10d, bits	0.7000	0.9385	0.41 ± 0.03	0.99 ± 0.00	0.22 ± 0.03	1.00 ± 0.00	0.99 ± 0.00	0.28 ± 0.02	0.40 ± 0.02	0.22 ± 0.03	0.29 ± 0.02	0.17 ± 0.01	97.53 ± 24.41
RandomForestClassifier23t15d, raw	0.7000	0.9366	0.45 ± 0.02	0.99 ± 0.00	0.21 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.28 ± 0.01	0.44 ± 0.02	0.21 ± 0.02	0.30 ± 0.01	0.17 ± 0.01	88.74 ± 19.73
RandomForestClassifier23t15d, octets	0.7000	0.9159	0.49 ± 0.02	1.00 ± 0.00	0.31 ± 0.03	1.00 ± 0.00	0.99 ± 0.00	0.38 ± 0.03	0.49 ± 0.02	0.31 ± 0.03	0.39 ± 0.03	0.24 ± 0.02	169.57 ± 50.79
RandomForestClassifier23t15d, bits	0.7000	0.8978	0.54 ± 0.04	1.00 ± 0.00	0.27 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.36 ± 0.03	0.54 ± 0.04	0.27 ± 0.03	0.38 ± 0.03	0.22 ± 0.02	174.02 ± 43.03
RandomForestClassifier23t20d, raw	0.7000	0.9451	0.43 ± 0.02	0.99 ± 0.00	0.26 ± 0.03	1.00 ± 0.00	0.99 ± 0.00	0.32 ± 0.01	0.42 ± 0.02	0.26 ± 0.03	0.33 ± 0.01	0.19 ± 0.01	95.17 ± 22.40
RandomForestClassifier23t20d, octets	0.7000	0.9302	0.46 ± 0.01	1.00 ± 0.00	0.38 ± 0.04	1.00 ± 0.00	0.99 ± 0.00	0.42 ± 0.02	0.45 ± 0.01	0.38 ± 0.04	0.41 ± 0.02	0.26 ± 0.02	176.16 ± 44.42
RandomForestClassifier23t20d, bits	0.7000	0.9234	0.48 ± 0.01	1.00 ± 0.00	0.36 ± 0.03	1.00 ± 0.00	0.99 ± 0.00	0.41 ± 0.02	0.47 ± 0.01	0.35 ± 0.03	0.41 ± 0.02	0.25 ± 0.02	176.06 ± 47.56
RandomForestClassifier23t25d, raw	0.7000	0.9618	0.42 ± 0.03	0.99 ± 0.00	0.27 ± 0.03	0.99 ± 0.00	0.98 ± 0.01	0.33 ± 0.02	0.40 ± 0.02	0.26 ± 0.03	0.33 ± 0.01	0.19 ± 0.01	64.69 ± 17.17
RandomForestClassifier23t25d, octets	0.7000	0.9515	0.40 ± 0.02	1.00 ± 0.00	0.44 ± 0.03	1.00 ± 0.00	0.99 ± 0.00	0.42 ± 0.01	0.40 ± 0.02	0.43 ± 0.03	0.42 ± 0.01	0.27 ± 0.00	143.97 ± 32.00
RandomForestClassifier23t25d, bits	0.7000	0.9600	0.37 ± 0.02	0.99 ± 0.00	0.45 ± 0.02	0.99 ± 0.00	0.99 ± 0.00	0.41 ± 0.01	0.37 ± 0.02	0.44 ± 0.03	0.40 ± 0.01	0.26 ± 0.01	126.13 ± 42.67
RandomForestClassifier23t30d, raw	0.7000	0.9768	0.41 ± 0.03	0.98 ± 0.01	0.30 ± 0.02	0.99 ± 0.00	0.97 ± 0.01	0.34 ± 0.01	0.39 ± 0.02	0.29 ± 0.02	0.33 ± 0.01	0.21 ± 0.01	40.38 ± 10.95
RandomForestClassifier23t30d, octets	0.7000	0.9679	0.37 ± 0.02	0.99 ± 0.00	0.47 ± 0.02	0.99 ± 0.00	0.99 ± 0.00	0.41 ± 0.01	0.36 ± 0.02	0.46 ± 0.02	0.41 ± 0.01	0.26 ± 0.00	102.08 ± 27.70
RandomForestClassifier23t30d, bits	0.7000	0.9801	0.32 ± 0.01	0.99 ± 0.00	0.49 ± 0.03	0.98 ± 0.00	0.98 ± 0.01	0.39 ± 0.01	0.31 ± 0.01	0.48 ± 0.03	0.38 ± 0.01	0.24 ± 0.01	69.26 ± 14.03
RandomForestClassifier23t, raw	0.7000	0.9908	0.39 ± 0.02	0.96 ± 0.00	0.38 ± 0.04	0.96 ± 0.00	0.93 ± 0.01	0.39 ± 0.02	0.36 ± 0.01	0.35 ± 0.03	0.35 ± 0.02	0.24 ± 0.02	17.59 ± 2.15
RandomForestClassifier23t, octets	0.7000	0.9827	0.33 ± 0.02	0.99 ± 0.00	0.47 ± 0.07	0.98 ± 0.01	0.97 ± 0.01	0.38 ± 0.03	0.32 ± 0.02	0.45 ± 0.07	0.38 ± 0.03	0.24 ± 0.03	52.85 ± 16.78
RandomForestClassifier23t, bits	0.7000	0.9843	0.31 ± 0.02	0.99 ± 0.00	0.51 ± 0.04	0.98 ± 0.01	0.97 ± 0.01	0.38 ± 0.02	0.30 ± 0.02	0.49 ± 0.04	0.38 ± 0.02	0.24 ± 0.02	56.13 ± 12.68
RandomForestClassifier, raw	0.7000	0.9908	0.40 ± 0.02	0.96 ± 0.00	0.38 ± 0.04	0.96 ± 0.00	0.93 ± 0.00	0.39 ± 0.03	0.36 ± 0.02	0.34 ± 0.04	0.35 ± 0.02	0.24 ± 0.02	17.16 ± 1.55
RandomForestClassifier, octets	0.7000	0.9832	0.32 ± 0.02	0.99 ± 0.00	0.48 ± 0.07	0.98 ± 0.01	0.97 ± 0.01	0.38 ± 0.03	0.31 ± 0.02	0.46 ± 0.07	0.38 ± 0.03	0.24 ± 0.02	53.32 ± 13.67
RandomForestClassifier, bits	0.7000	0.9850	0.30 ± 0.02	0.99 ± 0.00	0.53 ± 0.04	0.98 ± 0.00	0.97 ± 0.01	0.38 ± 0.02	0.29 ± 0.02	0.50 ± 0.04	0.38 ± 0.02	0.23 ± 0.02	56.61 ± 10.56

Figure 5.33: Performance metrics for RandomForestClassifier for 70% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
ExtraTreesClassifier23t10d, raw	0.7000	0.9542	0.46 ± 0.03	0.93 ± 0.01	0.06 ± 0.00	1.00 ± 0.00	0.93 ± 0.01	0.10 ± 0.00	0.39 ± 0.02	0.05 ± 0.00	0.14 ± 0.00	0.05 ± 0.00	16.87 ± 2.95
ExtraTreesClassifier23t10d, octets	0.7000	0.9593	0.43 ± 0.02	0.96 ± 0.01	0.11 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.17 ± 0.01	0.39 ± 0.02	0.10 ± 0.01	0.20 ± 0.01	0.09 ± 0.01	29.53 ± 6.14
ExtraTreesClassifier23t10d, bits	0.7000	0.9382	0.41 ± 0.03	0.99 ± 0.00	0.22 ± 0.03	1.00 ± 0.00	0.99 ± 0.01	0.28 ± 0.03	0.40 ± 0.03	0.21 ± 0.03	0.29 ± 0.02	0.17 ± 0.02	97.56 ± 27.87
ExtraTreesClassifier23t15d, raw	0.7000	0.9481	0.42 ± 0.02	0.96 ± 0.00	0.08 ± 0.01	1.00 ± 0.00	0.96 ± 0.00	0.14 ± 0.01	0.38 ± 0.02	0.08 ± 0.01	0.17 ± 0.01	0.07 ± 0.01	29.03 ± 4.34
ExtraTreesClassifier23t15d, octets	0.7000	0.9394	0.42 ± 0.02	0.99 ± 0.00	0.21 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.28 ± 0.02	0.41 ± 0.02	0.20 ± 0.02	0.29 ± 0.01	0.16 ± 0.01	89.42 ± 21.58
ExtraTreesClassifier23t15d, bits	0.7000	0.8846	0.56 ± 0.03	1.00 ± 0.00	0.26 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.35 ± 0.02	0.55 ± 0.03	0.26 ± 0.02	0.38 ± 0.02	0.22 ± 0.02	192.32 ± 45.08
ExtraTreesClassifier23t20d, raw	0.7000	0.9426	0.44 ± 0.03	0.98 ± 0.00	0.12 ± 0.00	1.00 ± 0.00	0.98 ± 0.00	0.19 ± 0.00	0.42 ± 0.03	0.12 ± 0.00	0.22 ± 0.01	0.10 ± 0.00	45.12 ± 6.45
ExtraTreesClassifier23t20d, octets	0.7000	0.9199	0.47 ± 0.01	0.99 ± 0.00	0.26 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.33 ± 0.02	0.47 ± 0.01	0.26 ± 0.02	0.35 ± 0.01	0.20 ± 0.01	133.82 ± 27.20
ExtraTreesClassifier23t20d, bits	0.7000	0.9008	0.54 ± 0.03	1.00 ± 0.00	0.33 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.41 ± 0.02	0.54 ± 0.03	0.32 ± 0.02	0.42 ± 0.02	0.25 ± 0.01	198.72 ± 44.75
ExtraTreesClassifier23t25d, raw	0.7000	0.9388	0.44 ± 0.02	0.98 ± 0.00	0.16 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.23 ± 0.01	0.43 ± 0.02	0.15 ± 0.01	0.25 ± 0.00	0.13 ± 0.00	63.92 ± 11.39
ExtraTreesClassifier23t25d, octets	0.7000	0.9123	0.51 ± 0.03	1.00 ± 0.00	0.29 ± 0.01	1.00 ± 0.00	0.99 ± 0.00	0.37 ± 0.01	0.50 ± 0.03	0.29 ± 0.01	0.38 ± 0.01	0.23 ± 0.01	160.56 ± 34.74
ExtraTreesClassifier23t25d, bits	0.7000	0.9437	0.42 ± 0.01	1.00 ± 0.00	0.40 ± 0.02	1.00 ± 0.00	0.99 ± 0.00	0.41 ± 0.01	0.42 ± 0.01	0.40 ± 0.02	0.41 ± 0.01	0.26 ± 0.01	151.70 ± 43.06
ExtraTreesClassifier23t30d, raw	0.7000	0.9433	0.43 ± 0.01	0.99 ± 0.00	0.19 ± 0.01	1.00 ± 0.00	0.98 ± 0.00	0.26 ± 0.01	0.42 ± 0.01	0.19 ± 0.01	0.28 ± 0.01	0.15 ± 0.01	71.00 ± 16.52
ExtraTreesClassifier23t30d, octets	0.7000	0.9344	0.45 ± 0.01	1.00 ± 0.00	0.35 ± 0.01	1.00 ± 0.00	0.99 ± 0.00	0.39 ± 0.01	0.44 ± 0.01	0.34 ± 0.01	0.39 ± 0.01	0.24 ± 0.01	148.95 ± 35.80
ExtraTreesClassifier23t30d, bits	0.7000	0.9707	0.36 ± 0.01	0.99 ± 0.00	0.44 ± 0.02	0.99 ± 0.00	0.98 ± 0.00	0.40 ± 0.01	0.35 ± 0.01	0.43 ± 0.02	0.39 ± 0.01	0.25 ± 0.01	87.65 ± 19.52
ExtraTreesClassifier23t, raw	0.7000	0.9909	0.40 ± 0.02	0.95 ± 0.00	0.35 ± 0.03	0.96 ± 0.00	0.92 ± 0.00	0.38 ± 0.02	0.36 ± 0.01	0.32 ± 0.03	0.34 ± 0.02	0.23 ± 0.01	15.63 ± 1.59
ExtraTreesClassifier23t, octets	0.7000	0.9809	0.35 ± 0.02	0.99 ± 0.00	0.41 ± 0.06	0.98 ± 0.00	0.97 ± 0.01	0.38 ± 0.03	0.34 ± 0.01	0.39 ± 0.05	0.36 ± 0.03	0.23 ± 0.02	49.23 ± 11.00
ExtraTreesClassifier23t, bits	0.7000	0.9793	0.33 ± 0.01	0.99 ± 0.00	0.45 ± 0.03	0.98 ± 0.00	0.98 ± 0.01	0.38 ± 0.01	0.33 ± 0.01	0.44 ± 0.03	0.38 ± 0.01	0.24 ± 0.01	62.51 ± 11.21
ExtraTreesClassifier, raw	0.7000	0.9912	0.40 ± 0.01	0.95 ± 0.00	0.36 ± 0.03	0.96 ± 0.00	0.92 ± 0.00	0.38 ± 0.02	0.36 ± 0.01	0.32 ± 0.03	0.34 ± 0.02	0.23 ± 0.02	15.41 ± 1.30
ExtraTreesClassifier, octets	0.7000	0.9811	0.34 ± 0.02	0.99 ± 0.00	0.42 ± 0.06	0.98 ± 0.00	0.97 ± 0.01	0.37 ± 0.03	0.33 ± 0.02	0.40 ± 0.05	0.36 ± 0.03	0.23 ± 0.02	51.22 ± 10.33
ExtraTreesClassifier, bits	0.7000	0.9795	0.32 ± 0.01	0.99 ± 0.00	0.47 ± 0.03	0.98 ± 0.00	0.98 ± 0.00	0.38 ± 0.01	0.32 ± 0.01	0.45 ± 0.03	0.38 ± 0.01	0.24 ± 0.01	65.54 ± 10.56

Figure 5.34: Performance metrics for ExtraTreesClassifier for 70% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
DecisionTreeClassifier10d, raw	0.8000	0.9634	0.57 ± 0.02	0.95 ± 0.00	0.12 ± 0.00	0.99 ± 0.00	0.94 ± 0.00	0.19 ± 0.00	0.52 ± 0.02	0.11 ± 0.00	0.24 ± 0.00	0.11 ± 0.00	19.13 ± 0.81
DecisionTreeClassifier10d, octets	0.8000	0.9656	0.55 ± 0.02	0.95 ± 0.01	0.12 ± 0.01	0.99 ± 0.00	0.95 ± 0.01	0.20 ± 0.01	0.50 ± 0.02	0.12 ± 0.01	0.24 ± 0.01	0.11 ± 0.01	19.91 ± 3.50
DecisionTreeClassifier10d, bits	0.8000	0.9504	0.57 ± 0.03	0.95 ± 0.01	0.08 ± 0.00	1.00 ± 0.00	0.95 ± 0.01	0.14 ± 0.01	0.52 ± 0.03	0.08 ± 0.00	0.20 ± 0.01	0.07 ± 0.00	18.75 ± 2.59
DecisionTreeClassifier15d, raw	0.8000	0.9541	0.58 ± 0.02	0.97 ± 0.01	0.13 ± 0.01	1.00 ± 0.00	0.96 ± 0.00	0.22 ± 0.02	0.54 ± 0.02	0.13 ± 0.01	0.27 ± 0.02	0.12 ± 0.01	28.83 ± 4.68
DecisionTreeClassifier15d, octets	0.8000	0.9547	0.57 ± 0.03	0.97 ± 0.01	0.15 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.24 ± 0.01	0.54 ± 0.03	0.15 ± 0.01	0.28 ± 0.01	0.14 ± 0.01	32.48 ± 8.01
DecisionTreeClassifier15d, bits	0.8000	0.9351	0.61 ± 0.02	0.98 ± 0.00	0.14 ± 0.01	1.00 ± 0.00	0.98 ± 0.01	0.22 ± 0.01	0.59 ± 0.02	0.13 ± 0.01	0.28 ± 0.01	0.12 ± 0.01	41.59 ± 7.36
DecisionTreeClassifier20d, raw	0.8000	0.9662	0.60 ± 0.01	0.96 ± 0.01	0.18 ± 0.02	0.99 ± 0.00	0.96 ± 0.01	0.28 ± 0.02	0.56 ± 0.01	0.18 ± 0.02	0.31 ± 0.02	0.16 ± 0.02	25.14 ± 5.10
DecisionTreeClassifier20d, octets	0.8000	0.9596	0.58 ± 0.02	0.98 ± 0.00	0.19 ± 0.01	1.00 ± 0.00	0.97 ± 0.00	0.29 ± 0.02	0.55 ± 0.02	0.19 ± 0.01	0.32 ± 0.02	0.17 ± 0.01	34.00 ± 4.61
DecisionTreeClassifier20d, bits	0.8000	0.9595	0.57 ± 0.01	0.98 ± 0.01	0.21 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.31 ± 0.01	0.55 ± 0.01	0.20 ± 0.01	0.34 ± 0.01	0.18 ± 0.01	37.42 ± 7.29
DecisionTreeClassifier25d, raw	0.8000	0.9823	0.58 ± 0.03	0.95 ± 0.01	0.23 ± 0.04	0.99 ± 0.00	0.94 ± 0.01	0.33 ± 0.04	0.52 ± 0.02	0.22 ± 0.03	0.34 ± 0.03	0.20 ± 0.03	15.17 ± 2.96
DecisionTreeClassifier25d, octets	0.8000	0.9741	0.56 ± 0.02	0.97 ± 0.01	0.25 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.35 ± 0.01	0.53 ± 0.01	0.25 ± 0.01	0.36 ± 0.01	0.21 ± 0.01	27.49 ± 5.51
DecisionTreeClassifier25d, bits	0.8000	0.9837	0.52 ± 0.03	0.96 ± 0.01	0.29 ± 0.03	0.98 ± 0.00	0.95 ± 0.01	0.37 ± 0.03	0.48 ± 0.02	0.27 ± 0.03	0.36 ± 0.02	0.23 ± 0.02	19.44 ± 4.51
DecisionTreeClassifier30d, raw	0.8000	0.9895	0.58 ± 0.03	0.93 ± 0.01	0.30 ± 0.04	0.98 ± 0.00	0.91 ± 0.01	0.39 ± 0.04	0.51 ± 0.02	0.28 ± 0.03	0.38 ± 0.03	0.25 ± 0.03	10.73 ± 1.44
DecisionTreeClassifier30d, octets	0.8000	0.9844	0.54 ± 0.02	0.96 ± 0.01	0.29 ± 0.03	0.98 ± 0.00	0.94 ± 0.01	0.38 ± 0.03	0.50 ± 0.01	0.27 ± 0.03	0.37 ± 0.02	0.23 ± 0.02	17.65 ± 3.06
DecisionTreeClassifier30d, bits	0.8000	0.9921	0.52 ± 0.03	0.94 ± 0.01	0.37 ± 0.05	0.96 ± 0.01	0.91 ± 0.02	0.43 ± 0.04	0.45 ± 0.02	0.34 ± 0.04	0.39 ± 0.03	0.28 ± 0.04	10.93 ± 2.30
DecisionTreeClassifier, raw	0.8000	0.9952	0.58 ± 0.02	0.90 ± 0.00	0.43 ± 0.05	0.95 ± 0.01	0.87 ± 0.00	0.49 ± 0.04	0.48 ± 0.02	0.37 ± 0.04	0.42 ± 0.03	0.33 ± 0.03	6.77 ± 0.49
DecisionTreeClassifier, octets	0.8000	0.9929	0.53 ± 0.03	0.93 ± 0.01	0.40 ± 0.05	0.96 ± 0.01	0.90 ± 0.02	0.46 ± 0.05	0.46 ± 0.02	0.36 ± 0.04	0.41 ± 0.03	0.30 ± 0.04	10.05 ± 2.00
DecisionTreeClassifier, bits	0.8000	0.9950	0.57 ± 0.05	0.91 ± 0.02	0.44 ± 0.07	0.94 ± 0.01	0.87 ± 0.02	0.50 ± 0.06	0.48 ± 0.03	0.39 ± 0.05	0.43 ± 0.04	0.33 ± 0.05	7.29 ± 1.74

Figure 5.35: Performance metrics for DecisionTreeClassifier for 80% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
RandomForestClassifier23t10d, raw	0.8000	0.9640	0.59 ± 0.03	0.94 ± 0.01	0.11 ± 0.01	1.00 ± 0.00	0.94 ± 0.01	0.18 ± 0.02	0.53 ± 0.02	0.10 ± 0.01	0.23 ± 0.01	0.10 ± 0.01	17.10 ± 2.69
RandomForestClassifier23t10d, octets	0.8000	0.9637	0.55 ± 0.03	0.96 ± 0.01	0.16 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.24 ± 0.01	0.51 ± 0.02	0.15 ± 0.01	0.28 ± 0.01	0.14 ± 0.01	28.11 ± 4.56
RandomForestClassifier23t10d, bits	0.8000	0.9624	0.54 ± 0.02	0.96 ± 0.01	0.15 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.24 ± 0.01	0.50 ± 0.02	0.15 ± 0.01	0.27 ± 0.01	0.13 ± 0.01	29.35 ± 5.25
RandomForestClassifier23t15d, raw	0.8000	0.9618	0.55 ± 0.02	0.97 ± 0.01	0.17 ± 0.01	1.00 ± 0.00	0.97 ± 0.01	0.26 ± 0.01	0.51 ± 0.01	0.16 ± 0.01	0.29 ± 0.01	0.15 ± 0.01	30.57 ± 4.41
RandomForestClassifier23t15d, octets	0.8000	0.9598	0.54 ± 0.02	0.98 ± 0.01	0.25 ± 0.02	1.00 ± 0.00	0.98 ± 0.01	0.34 ± 0.02	0.53 ± 0.02	0.24 ± 0.02	0.36 ± 0.01	0.20 ± 0.01	48.81 ± 9.98
RandomForestClassifier23t15d, bits	0.8000	0.9550	0.54 ± 0.03	0.98 ± 0.00	0.25 ± 0.02	1.00 ± 0.00	0.98 ± 0.00	0.34 ± 0.01	0.52 ± 0.02	0.25 ± 0.02	0.36 ± 0.01	0.20 ± 0.01	57.97 ± 12.08
RandomForestClassifier23t20d, raw	0.8000	0.9676	0.54 ± 0.02	0.97 ± 0.01	0.20 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.29 ± 0.01	0.51 ± 0.01	0.19 ± 0.01	0.31 ± 0.01	0.17 ± 0.01	31.11 ± 6.00
RandomForestClassifier23t20d, octets	0.8000	0.9640	0.52 ± 0.02	0.99 ± 0.00	0.32 ± 0.02	0.99 ± 0.00	0.98 ± 0.00	0.39 ± 0.01	0.51 ± 0.02	0.31 ± 0.02	0.40 ± 0.01	0.25 ± 0.01	55.63 ± 11.66
RandomForestClassifier23t20d, bits	0.8000	0.9645	0.51 ± 0.02	0.99 ± 0.00	0.32 ± 0.02	0.99 ± 0.00	0.98 ± 0.01	0.39 ± 0.01	0.50 ± 0.02	0.31 ± 0.02	0.39 ± 0.00	0.24 ± 0.00	55.04 ± 11.81
RandomForestClassifier23t25d, raw	0.8000	0.9791	0.52 ± 0.03	0.96 ± 0.01	0.24 ± 0.02	0.99 ± 0.00	0.95 ± 0.01	0.32 ± 0.02	0.48 ± 0.02	0.23 ± 0.02	0.33 ± 0.01	0.19 ± 0.01	21.47 ± 4.01
RandomForestClassifier23t25d, octets	0.8000	0.9764	0.49 ± 0.03	0.98 ± 0.01	0.38 ± 0.02	0.99 ± 0.00	0.97 ± 0.01	0.43 ± 0.01	0.47 ± 0.02	0.37 ± 0.02	0.42 ± 0.01	0.27 ± 0.01	44.01 ± 10.33
RandomForestClassifier23t25d, bits	0.8000	0.9828	0.45 ± 0.03	0.98 ± 0.01	0.40 ± 0.02	0.98 ± 0.00	0.97 ± 0.01	0.42 ± 0.02	0.43 ± 0.02	0.39 ± 0.02	0.41 ± 0.01	0.27 ± 0.01	34.14 ± 6.72
RandomForestClassifier23t30d, raw	0.8000	0.9873	0.52 ± 0.03	0.95 ± 0.01	0.30 ± 0.03	0.98 ± 0.00	0.93 ± 0.01	0.38 ± 0.03	0.47 ± 0.03	0.28 ± 0.02	0.36 ± 0.02	0.23 ± 0.02	15.00 ± 2.40
RandomForestClassifier23t30d, octets	0.8000	0.9849	0.46 ± 0.03	0.98 ± 0.01	0.41 ± 0.05	0.98 ± 0.00	0.96 ± 0.01	0.43 ± 0.03	0.43 ± 0.03	0.40 ± 0.05	0.41 ± 0.03	0.28 ± 0.03	29.18 ± 6.14
RandomForestClassifier23t30d, bits	0.8000	0.9902	0.42 ± 0.04	0.98 ± 0.01	0.50 ± 0.03	0.97 ± 0.01	0.95 ± 0.01	0.45 ± 0.03	0.39 ± 0.04	0.46 ± 0.03	0.43 ± 0.03	0.29 ± 0.03	23.75 ± 6.11
RandomForestClassifier23t, raw	0.8000	0.9959	0.57 ± 0.03	0.91 ± 0.01	0.50 ± 0.04	0.94 ± 0.01	0.87 ± 0.01	0.53 ± 0.03	0.49 ± 0.02	0.43 ± 0.03	0.46 ± 0.02	0.36 ± 0.03	6.83 ± 0.66
RandomForestClassifier23t, octets	0.8000	0.9924	0.44 ± 0.04	0.97 ± 0.01	0.51 ± 0.05	0.96 ± 0.01	0.93 ± 0.01	0.47 ± 0.04	0.41 ± 0.04	0.47 ± 0.05	0.44 ± 0.03	0.31 ± 0.03	17.01 ± 3.71
RandomForestClassifier23t, bits	0.8000	0.9927	0.43 ± 0.05	0.97 ± 0.01	0.53 ± 0.04	0.95 ± 0.01	0.93 ± 0.01	0.47 ± 0.04	0.40 ± 0.04	0.49 ± 0.04	0.44 ± 0.03	0.31 ± 0.04	17.89 ± 4.34
RandomForestClassifier, raw	0.8000	0.9959	0.58 ± 0.03	0.91 ± 0.01	0.50 ± 0.05	0.94 ± 0.01	0.87 ± 0.01	0.53 ± 0.03	0.49 ± 0.02	0.43 ± 0.04	0.46 ± 0.03	0.37 ± 0.03	6.80 ± 0.70
RandomForestClassifier, octets	0.8000	0.9922	0.43 ± 0.05	0.97 ± 0.01	0.51 ± 0.06	0.96 ± 0.01	0.93 ± 0.01	0.47 ± 0.05	0.40 ± 0.04	0.47 ± 0.06	0.44 ± 0.04	0.31 ± 0.04	18.16 ± 4.27
RandomForestClassifier, bits	0.8000	0.9925	0.41 ± 0.05	0.97 ± 0.01	0.55 ± 0.04	0.96 ± 0.01	0.93 ± 0.01	0.47 ± 0.04	0.38 ± 0.05	0.51 ± 0.03	0.44 ± 0.04	0.31 ± 0.04	20.14 ± 5.30

Figure 5.36: Performance metrics for RandomForestClassifier for 80% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
ExtraTreesClassifier23t10d, raw	0.8000	0.9653	0.60 ± 0.02	0.88 ± 0.01	0.06 ± 0.00	0.99 ± 0.00	0.88 ± 0.01	0.11 ± 0.00	0.49 ± 0.01	0.06 ± 0.00	0.16 ± 0.00	0.06 ± 0.00	8.79 ± 0.58
ExtraTreesClassifier23t10d, octets	0.8000	0.9675	0.59 ± 0.02	0.92 ± 0.01	0.09 ± 0.01	0.99 ± 0.00	0.91 ± 0.01	0.16 ± 0.01	0.51 ± 0.01	0.08 ± 0.01	0.21 ± 0.01	0.08 ± 0.01	13.87 ± 1.24
ExtraTreesClassifier23t10d, bits	0.8000	0.9615	0.53 ± 0.03	0.97 ± 0.01	0.15 ± 0.02	0.99 ± 0.00	0.96 ± 0.01	0.24 ± 0.02	0.50 ± 0.02	0.15 ± 0.02	0.27 ± 0.01	0.13 ± 0.01	30.01 ± 5.58
ExtraTreesClassifier23t15d, raw	0.8000	0.9621	0.58 ± 0.01	0.93 ± 0.00	0.08 ± 0.00	1.00 ± 0.00	0.92 ± 0.00	0.14 ± 0.01	0.51 ± 0.01	0.08 ± 0.00	0.20 ± 0.01	0.08 ± 0.00	13.84 ± 0.64
ExtraTreesClassifier23t15d, octets	0.8000	0.9617	0.53 ± 0.02	0.97 ± 0.01	0.15 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.24 ± 0.01	0.50 ± 0.02	0.15 ± 0.01	0.27 ± 0.01	0.14 ± 0.01	29.65 ± 3.79
ExtraTreesClassifier23t15d, bits	0.8000	0.9503	0.54 ± 0.03	0.99 ± 0.00	0.25 ± 0.02	1.00 ± 0.00	0.98 ± 0.00	0.34 ± 0.01	0.53 ± 0.03	0.24 ± 0.02	0.36 ± 0.01	0.20 ± 0.01	62.91 ± 13.62
ExtraTreesClassifier23t20d, raw	0.8000	0.9609	0.56 ± 0.02	0.96 ± 0.00	0.11 ± 0.00	1.00 ± 0.00	0.95 ± 0.00	0.19 ± 0.01	0.51 ± 0.01	0.11 ± 0.00	0.24 ± 0.00	0.11 ± 0.00	20.62 ± 1.21
ExtraTreesClassifier23t20d, octets	0.8000	0.9594	0.53 ± 0.02	0.98 ± 0.01	0.22 ± 0.01	1.00 ± 0.00	0.98 ± 0.01	0.31 ± 0.01	0.51 ± 0.02	0.22 ± 0.01	0.33 ± 0.01	0.18 ± 0.01	45.16 ± 7.54
ExtraTreesClassifier23t20d, bits	0.8000	0.9552	0.54 ± 0.02	0.99 ± 0.00	0.30 ± 0.02	1.00 ± 0.00	0.98 ± 0.00	0.38 ± 0.01	0.53 ± 0.02	0.29 ± 0.02	0.39 ± 0.00	0.24 ± 0.01	66.35 ± 14.62
ExtraTreesClassifier23t25d, raw	0.8000	0.9623	0.54 ± 0.02	0.96 ± 0.00	0.15 ± 0.01	0.99 ± 0.00	0.96 ± 0.01	0.23 ± 0.01	0.51 ± 0.01	0.14 ± 0.01	0.27 ± 0.00	0.13 ± 0.00	25.75 ± 2.55
ExtraTreesClassifier23t25d, octets	0.8000	0.9625	0.53 ± 0.02	0.98 ± 0.00	0.28 ± 0.02	0.99 ± 0.00	0.98 ± 0.01	0.36 ± 0.01	0.51 ± 0.02	0.27 ± 0.02	0.37 ± 0.01	0.22 ± 0.01	50.31 ± 10.16
ExtraTreesClassifier23t25d, bits	0.8000	0.9745	0.48 ± 0.02	0.98 ± 0.00	0.36 ± 0.02	0.99 ± 0.00	0.98 ± 0.01	0.41 ± 0.01	0.47 ± 0.02	0.35 ± 0.02	0.41 ± 0.00	0.26 ± 0.01	45.25 ± 9.64
ExtraTreesClassifier23t30d, raw	0.8000	0.9690	0.52 ± 0.02	0.97 ± 0.01	0.18 ± 0.00	0.99 ± 0.00	0.96 ± 0.01	0.27 ± 0.00	0.49 ± 0.01	0.17 ± 0.00	0.29 ± 0.00	0.15 ± 0.00	26.48 ± 4.62
ExtraTreesClassifier23t30d, octets	0.8000	0.9726	0.49 ± 0.02	0.98 ± 0.01	0.33 ± 0.02	0.99 ± 0.00	0.97 ± 0.01	0.39 ± 0.01	0.48 ± 0.02	0.32 ± 0.02	0.39 ± 0.00	0.24 ± 0.01	43.85 ± 10.62
ExtraTreesClassifier23t30d, bits	0.8000	0.9870	0.44 ± 0.03	0.98 ± 0.01	0.44 ± 0.04	0.98 ± 0.01	0.96 ± 0.01	0.44 ± 0.03	0.41 ± 0.03	0.42 ± 0.03	0.41 ± 0.02	0.28 ± 0.02	27.38 ± 6.37
ExtraTreesClassifier23t, raw	0.8000	0.9957	0.58 ± 0.02	0.91 ± 0.01	0.46 ± 0.04	0.94 ± 0.01	0.87 ± 0.00	0.51 ± 0.03	0.48 ± 0.02	0.40 ± 0.03	0.44 ± 0.02	0.35 ± 0.03	6.60 ± 0.47
ExtraTreesClassifier23t, octets	0.8000	0.9915	0.46 ± 0.04	0.96 ± 0.01	0.46 ± 0.05	0.96 ± 0.01	0.93 ± 0.01	0.46 ± 0.04	0.42 ± 0.03	0.42 ± 0.04	0.42 ± 0.03	0.30 ± 0.03	16.63 ± 3.76
ExtraTreesClassifier23t, bits	0.8000	0.9903	0.43 ± 0.05	0.97 ± 0.01	0.47 ± 0.04	0.97 ± 0.01	0.94 ± 0.01	0.45 ± 0.04	0.40 ± 0.04	0.44 ± 0.04	0.42 ± 0.03	0.29 ± 0.03	21.85 ± 5.35
ExtraTreesClassifier, raw	0.8000	0.9959	0.59 ± 0.02	0.90 ± 0.01	0.47 ± 0.04	0.94 ± 0.01	0.86 ± 0.00	0.52 ± 0.03	0.49 ± 0.02	0.41 ± 0.03	0.45 ± 0.02	0.36 ± 0.03	6.38 ± 0.45
ExtraTreesClassifier, octets	0.8000	0.9912	0.44 ± 0.04	0.97 ± 0.01	0.46 ± 0.05	0.96 ± 0.01	0.93 ± 0.01	0.45 ± 0.04	0.41 ± 0.03	0.43 ± 0.04	0.42 ± 0.03	0.29 ± 0.03	18.05 ± 3.41
ExtraTreesClassifier, bits	0.8000	0.9900	0.41 ± 0.04	0.98 ± 0.01	0.49 ± 0.04	0.97 ± 0.01	0.95 ± 0.01	0.44 ± 0.04	0.39 ± 0.03	0.45 ± 0.04	0.42 ± 0.03	0.29 ± 0.03	23.88 ± 5.27

Figure 5.37: Performance metrics for ExtraTreesClassifier for 80% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
DecisionTreeClassifier10d, raw	0.9000	0.9822	0.65 ± 0.02	0.88 ± 0.01	0.13 ± 0.01	0.99 ± 0.00	0.88 ± 0.01	0.22 ± 0.02	0.53 ± 0.01	0.12 ± 0.01	0.25 ± 0.01	0.12 ± 0.01	7.81 ± 0.60
DecisionTreeClassifier10d, octets	0.9000	0.9831	0.64 ± 0.02	0.89 ± 0.01	0.14 ± 0.02	0.99 ± 0.00	0.88 ± 0.01	0.23 ± 0.02	0.53 ± 0.02	0.13 ± 0.01	0.26 ± 0.01	0.13 ± 0.01	8.16 ± 0.71
DecisionTreeClassifier10d, bits	0.9000	0.9806	0.64 ± 0.01	0.89 ± 0.01	0.12 ± 0.01	0.99 ± 0.00	0.88 ± 0.01	0.21 ± 0.01	0.53 ± 0.00	0.11 ± 0.01	0.24 ± 0.01	0.12 ± 0.01	8.12 ± 0.80
DecisionTreeClassifier15d, raw	0.9000	0.9806	0.66 ± 0.01	0.92 ± 0.01	0.17 ± 0.02	0.99 ± 0.00	0.91 ± 0.01	0.26 ± 0.02	0.57 ± 0.01	0.16 ± 0.01	0.30 ± 0.01	0.15 ± 0.01	10.67 ± 1.16
DecisionTreeClassifier15d, octets	0.9000	0.9820	0.65 ± 0.01	0.92 ± 0.01	0.19 ± 0.02	0.99 ± 0.00	0.91 ± 0.01	0.29 ± 0.02	0.58 ± 0.02	0.18 ± 0.02	0.32 ± 0.01	0.17 ± 0.02	11.07 ± 1.09
DecisionTreeClassifier15d, bits	0.9000	0.9812	0.65 ± 0.01	0.93 ± 0.00	0.20 ± 0.01	0.99 ± 0.00	0.93 ± 0.00	0.31 ± 0.02	0.59 ± 0.01	0.19 ± 0.01	0.33 ± 0.01	0.18 ± 0.01	12.63 ± 0.67
DecisionTreeClassifier20d, raw	0.9000	0.9884	0.69 ± 0.04	0.90 ± 0.01	0.25 ± 0.03	0.98 ± 0.00	0.89 ± 0.01	0.36 ± 0.04	0.59 ± 0.02	0.23 ± 0.03	0.37 ± 0.03	0.22 ± 0.03	8.45 ± 1.34
DecisionTreeClassifier20d, octets	0.9000	0.9857	0.67 ± 0.02	0.93 ± 0.00	0.24 ± 0.02	0.99 ± 0.00	0.92 ± 0.01	0.36 ± 0.02	0.59 ± 0.02	0.23 ± 0.02	0.37 ± 0.01	0.22 ± 0.01	11.00 ± 0.71
DecisionTreeClassifier20d, bits	0.9000	0.9875	0.67 ± 0.02	0.93 ± 0.01	0.29 ± 0.02	0.99 ± 0.00	0.92 ± 0.01	0.40 ± 0.02	0.60 ± 0.01	0.27 ± 0.02	0.40 ± 0.02	0.25 ± 0.02	11.09 ± 1.56
DecisionTreeClassifier25d, raw	0.9000	0.9930	0.73 ± 0.04	0.88 ± 0.02	0.35 ± 0.04	0.98 ± 0.00	0.87 ± 0.01	0.47 ± 0.05	0.61 ± 0.02	0.32 ± 0.04	0.44 ± 0.03	0.31 ± 0.04	6.30 ± 1.01
DecisionTreeClassifier25d, octets	0.9000	0.9917	0.69 ± 0.02	0.91 ± 0.01	0.34 ± 0.03	0.98 ± 0.00	0.89 ± 0.01	0.45 ± 0.03	0.59 ± 0.02	0.31 ± 0.03	0.43 ± 0.02	0.29 ± 0.03	7.83 ± 0.91
DecisionTreeClassifier25d, bits	0.9000	0.9958	0.71 ± 0.05	0.88 ± 0.02	0.46 ± 0.05	0.95 ± 0.01	0.86 ± 0.02	0.56 ± 0.05	0.59 ± 0.02	0.42 ± 0.04	0.49 ± 0.03	0.39 ± 0.05	5.18 ± 1.02
DecisionTreeClassifier30d, raw	0.9000	0.9965	0.78 ± 0.02	0.85 ± 0.01	0.48 ± 0.06	0.96 ± 0.01	0.84 ± 0.00	0.59 ± 0.05	0.63 ± 0.02	0.44 ± 0.06	0.52 ± 0.04	0.42 ± 0.05	4.14 ± 0.44
DecisionTreeClassifier30d, octets	0.9000	0.9947	0.71 ± 0.03	0.89 ± 0.02	0.42 ± 0.04	0.96 ± 0.00	0.87 ± 0.01	0.52 ± 0.03	0.60 ± 0.02	0.38 ± 0.03	0.48 ± 0.02	0.36 ± 0.03	5.93 ± 0.84
DecisionTreeClassifier30d, bits	0.9000	0.9985	0.78 ± 0.02	0.80 ± 0.03	0.64 ± 0.06	0.88 ± 0.03	0.80 ± 0.01	0.70 ± 0.05	0.58 ± 0.02	0.53 ± 0.04	0.55 ± 0.02	0.54 ± 0.05	2.67 ± 0.47
DecisionTreeClassifier, raw	0.9000	0.9987	0.79 ± 0.02	0.79 ± 0.01	0.66 ± 0.05	0.88 ± 0.03	0.79 ± 0.01	0.72 ± 0.03	0.58 ± 0.02	0.53 ± 0.02	0.55 ± 0.01	0.56 ± 0.03	2.46 ± 0.23
DecisionTreeClassifier, octets	0.9000	0.9985	0.78 ± 0.01	0.80 ± 0.03	0.65 ± 0.06	0.88 ± 0.03	0.80 ± 0.01	0.71 ± 0.04	0.58 ± 0.02	0.53 ± 0.03	0.55 ± 0.01	0.55 ± 0.05	2.69 ± 0.42
DecisionTreeClassifier, bits	0.9000	0.9990	0.78 ± 0.01	0.75 ± 0.03	0.68 ± 0.06	0.82 ± 0.05	0.76 ± 0.02	0.73 ± 0.03	0.53 ± 0.03	0.51 ± 0.01	0.52 ± 0.01	0.57 ± 0.04	2.16 ± 0.35

Figure 5.38: Performance metrics for DecisionTreeClassifier for 90% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
RandomForestClassifier23t10d, raw	0.9000	0.9813	0.68 ± 0.02	0.87 ± 0.02	0.13 ± 0.03	0.99 ± 0.00	0.87 ± 0.02	0.22 ± 0.04	0.56 ± 0.02	0.12 ± 0.02	0.26 ± 0.02	0.12 ± 0.02	7.38 ± 1.22
RandomForestClassifier23t10d, octets	0.9000	0.9826	0.66 ± 0.02	0.89 ± 0.01	0.15 ± 0.01	0.99 ± 0.00	0.88 ± 0.01	0.24 ± 0.02	0.55 ± 0.00	0.13 ± 0.01	0.27 ± 0.01	0.14 ± 0.01	8.28 ± 1.10
RandomForestClassifier23t10d, bits	0.9000	0.9814	0.66 ± 0.01	0.89 ± 0.01	0.14 ± 0.01	0.99 ± 0.00	0.88 ± 0.01	0.23 ± 0.02	0.55 ± 0.01	0.13 ± 0.01	0.26 ± 0.01	0.13 ± 0.01	8.27 ± 1.23
RandomForestClassifier23t15d, raw	0.9000	0.9817	0.65 ± 0.02	0.91 ± 0.01	0.15 ± 0.01	0.99 ± 0.00	0.90 ± 0.01	0.25 ± 0.02	0.55 ± 0.01	0.14 ± 0.01	0.28 ± 0.01	0.14 ± 0.01	9.45 ± 0.76
RandomForestClassifier23t15d, octets	0.9000	0.9820	0.64 ± 0.02	0.93 ± 0.01	0.20 ± 0.01	0.99 ± 0.00	0.92 ± 0.01	0.30 ± 0.02	0.57 ± 0.01	0.18 ± 0.01	0.32 ± 0.01	0.18 ± 0.01	11.92 ± 1.42
RandomForestClassifier23t15d, bits	0.9000	0.9805	0.65 ± 0.01	0.94 ± 0.01	0.20 ± 0.01	0.99 ± 0.00	0.93 ± 0.01	0.31 ± 0.01	0.59 ± 0.01	0.20 ± 0.01	0.34 ± 0.00	0.18 ± 0.01	13.62 ± 1.58
RandomForestClassifier23t20d, raw	0.9000	0.9856	0.65 ± 0.02	0.91 ± 0.01	0.21 ± 0.02	0.99 ± 0.00	0.90 ± 0.01	0.31 ± 0.03	0.56 ± 0.01	0.19 ± 0.02	0.33 ± 0.02	0.19 ± 0.02	9.57 ± 1.30
RandomForestClassifier23t20d, octets	0.9000	0.9862	0.64 ± 0.02	0.94 ± 0.01	0.28 ± 0.02	0.99 ± 0.00	0.93 ± 0.01	0.39 ± 0.02	0.58 ± 0.02	0.27 ± 0.02	0.39 ± 0.02	0.24 ± 0.02	12.69 ± 1.80
RandomForestClassifier23t20d, bits	0.9000	0.9876	0.64 ± 0.03	0.94 ± 0.01	0.33 ± 0.02	0.98 ± 0.00	0.93 ± 0.01	0.43 ± 0.02	0.58 ± 0.02	0.31 ± 0.02	0.42 ± 0.01	0.27 ± 0.01	13.14 ± 1.93
RandomForestClassifier23t25d, raw	0.9000	0.9908	0.67 ± 0.04	0.90 ± 0.02	0.29 ± 0.03	0.98 ± 0.00	0.89 ± 0.02	0.41 ± 0.04	0.57 ± 0.03	0.27 ± 0.03	0.40 ± 0.03	0.26 ± 0.03	7.84 ± 1.37
RandomForestClassifier23t25d, octets	0.9000	0.9908	0.64 ± 0.04	0.93 ± 0.01	0.38 ± 0.03	0.98 ± 0.00	0.92 ± 0.01	0.47 ± 0.03	0.58 ± 0.03	0.35 ± 0.03	0.45 ± 0.03	0.31 ± 0.03	10.64 ± 1.64
RandomForestClassifier23t25d, bits	0.9000	0.9941	0.66 ± 0.04	0.92 ± 0.02	0.46 ± 0.04	0.97 ± 0.00	0.90 ± 0.01	0.55 ± 0.04	0.59 ± 0.03	0.43 ± 0.03	0.50 ± 0.03	0.38 ± 0.03	7.99 ± 1.54
RandomForestClassifier23t30d, raw	0.9000	0.9942	0.72 ± 0.06	0.88 ± 0.02	0.39 ± 0.04	0.97 ± 0.00	0.87 ± 0.02	0.51 ± 0.05	0.60 ± 0.03	0.36 ± 0.04	0.47 ± 0.04	0.34 ± 0.04	6.08 ± 1.34
RandomForestClassifier23t30d, octets	0.9000	0.9940	0.66 ± 0.03	0.92 ± 0.01	0.46 ± 0.04	0.97 ± 0.00	0.90 ± 0.01	0.54 ± 0.03	0.59 ± 0.02	0.43 ± 0.03	0.50 ± 0.03	0.37 ± 0.03	7.99 ± 1.20
RandomForestClassifier23t30d, bits	0.9000	0.9967	0.70 ± 0.05	0.90 ± 0.02	0.58 ± 0.05	0.94 ± 0.01	0.87 ± 0.01	0.63 ± 0.05	0.60 ± 0.04	0.52 ± 0.04	0.56 ± 0.04	0.46 ± 0.05	5.39 ± 1.10
RandomForestClassifier23t, raw	0.9000	0.9985	0.78 ± 0.01	0.83 ± 0.01	0.67 ± 0.05	0.89 ± 0.02	0.81 ± 0.01	0.72 ± 0.02	0.60 ± 0.01	0.56 ± 0.02	0.58 ± 0.01	0.56 ± 0.03	2.79 ± 0.21
RandomForestClassifier23t, octets	0.9000	0.9975	0.72 ± 0.04	0.89 ± 0.01	0.63 ± 0.07	0.93 ± 0.01	0.86 ± 0.01	0.67 ± 0.06	0.61 ± 0.03	0.55 ± 0.06	0.58 ± 0.04	0.51 ± 0.06	4.43 ± 0.89
RandomForestClassifier23t, bits	0.9000	0.9974	0.72 ± 0.05	0.89 ± 0.01	0.63 ± 0.06	0.93 ± 0.01	0.86 ± 0.01	0.67 ± 0.05	0.61 ± 0.03	0.56 ± 0.05	0.58 ± 0.04	0.50 ± 0.05	4.58 ± 1.00
RandomForestClassifier, raw	0.9000	0.9985	0.78 ± 0.01	0.82 ± 0.01	0.67 ± 0.05	0.89 ± 0.02	0.81 ± 0.01	0.72 ± 0.02	0.61 ± 0.01	0.56 ± 0.02	0.58 ± 0.01	0.56 ± 0.03	2.78 ± 0.18
RandomForestClassifier, octets	0.9000	0.9973	0.71 ± 0.05	0.90 ± 0.01	0.62 ± 0.07	0.93 ± 0.01	0.86 ± 0.01	0.66 ± 0.06	0.61 ± 0.04	0.55 ± 0.06	0.58 ± 0.05	0.50 ± 0.06	4.69 ± 1.15
RandomForestClassifier, bits	0.9000	0.9972	0.71 ± 0.04	0.90 ± 0.01	0.62 ± 0.05	0.93 ± 0.01	0.87 ± 0.01	0.66 ± 0.04	0.61 ± 0.03	0.55 ± 0.04	0.58 ± 0.03	0.50 ± 0.04	4.86 ± 0.89

Figure 5.39: Performance metrics for RandomForestClassifier for 90% resulting traffic coverage across all 5 folds.

	Resulting Average Traffic Coverage	Training Traffic Coverage	TPR (Recall)	TNR (Specificity)	PPV (Precision)	NPV	Accuracy	FScore	BM (Informed-ness)	MK (Marked-ness)	MCC	TS	Flow Operations Reduction
ExtraTreesClassifier23t10d, raw	0.9000	0.9869	0.69 ± 0.02	0.81 ± 0.01	0.13 ± 0.03	0.98 ± 0.01	0.81 ± 0.01	0.22 ± 0.05	0.50 ± 0.02	0.12 ± 0.03	0.24 ± 0.03	0.12 ± 0.03	4.82 ± 0.37
ExtraTreesClassifier23t10d, octets	0.9000	0.9837	0.70 ± 0.02	0.83 ± 0.01	0.11 ± 0.02	0.99 ± 0.00	0.83 ± 0.01	0.19 ± 0.03	0.53 ± 0.02	0.10 ± 0.02	0.23 ± 0.01	0.11 ± 0.02	5.41 ± 0.35
ExtraTreesClassifier23t10d, bits	0.9000	0.9806	0.67 ± 0.01	0.89 ± 0.01	0.13 ± 0.01	0.99 ± 0.00	0.88 ± 0.01	0.22 ± 0.02	0.56 ± 0.02	0.12 ± 0.01	0.26 ± 0.01	0.12 ± 0.01	8.34 ± 1.26
ExtraTreesClassifier23t15d, raw	0.9000	0.9784	0.72 ± 0.01	0.86 ± 0.01	0.10 ± 0.01	0.99 ± 0.00	0.85 ± 0.01	0.17 ± 0.01	0.58 ± 0.00	0.09 ± 0.01	0.23 ± 0.01	0.09 ± 0.01	6.56 ± 0.56
ExtraTreesClassifier23t15d, octets	0.9000	0.9785	0.68 ± 0.01	0.90 ± 0.01	0.13 ± 0.01	0.99 ± 0.00	0.90 ± 0.01	0.22 ± 0.01	0.58 ± 0.01	0.12 ± 0.01	0.26 ± 0.01	0.12 ± 0.01	9.25 ± 1.07
ExtraTreesClassifier23t15d, bits	0.9000	0.9783	0.66 ± 0.01	0.94 ± 0.01	0.20 ± 0.01	0.99 ± 0.00	0.94 ± 0.01	0.30 ± 0.01	0.60 ± 0.01	0.19 ± 0.01	0.34 ± 0.01	0.18 ± 0.01	14.80 ± 1.80
ExtraTreesClassifier23t20d, raw	0.9000	0.9789	0.68 ± 0.01	0.90 ± 0.01	0.13 ± 0.01	0.99 ± 0.00	0.89 ± 0.01	0.22 ± 0.01	0.58 ± 0.01	0.12 ± 0.01	0.26 ± 0.01	0.12 ± 0.01	8.80 ± 0.79
ExtraTreesClassifier23t20d, octets	0.9000	0.9802	0.66 ± 0.01	0.92 ± 0.01	0.18 ± 0.01	0.99 ± 0.00	0.92 ± 0.01	0.28 ± 0.01	0.58 ± 0.01	0.17 ± 0.01	0.31 ± 0.01	0.16 ± 0.01	11.66 ± 1.61
ExtraTreesClassifier23t20d, bits	0.9000	0.9847	0.64 ± 0.02	0.95 ± 0.01	0.30 ± 0.02	0.99 ± 0.00	0.94 ± 0.01	0.41 ± 0.02	0.59 ± 0.01	0.29 ± 0.02	0.41 ± 0.01	0.26 ± 0.01	15.03 ± 1.68
ExtraTreesClassifier23t25d, raw	0.9000	0.9827	0.66 ± 0.01	0.91 ± 0.01	0.17 ± 0.01	0.99 ± 0.00	0.90 ± 0.01	0.27 ± 0.01	0.57 ± 0.01	0.16 ± 0.01	0.30 ± 0.01	0.15 ± 0.01	9.47 ± 0.91
ExtraTreesClassifier23t25d, octets	0.9000	0.9846	0.64 ± 0.02	0.94 ± 0.01	0.25 ± 0.02	0.99 ± 0.00	0.93 ± 0.01	0.36 ± 0.02	0.58 ± 0.01	0.24 ± 0.02	0.37 ± 0.01	0.22 ± 0.02	12.58 ± 1.86
ExtraTreesClassifier23t25d, bits	0.9000	0.9922	0.65 ± 0.04	0.93 ± 0.01	0.41 ± 0.03	0.97 ± 0.00	0.91 ± 0.01	0.50 ± 0.03	0.58 ± 0.03	0.39 ± 0.03	0.47 ± 0.02	0.34 ± 0.03	9.57 ± 1.66
ExtraTreesClassifier23t30d, raw	0.9000	0.9884	0.65 ± 0.02	0.91 ± 0.01	0.24 ± 0.02	0.98 ± 0.00	0.90 ± 0.01	0.35 ± 0.03	0.56 ± 0.01	0.23 ± 0.02	0.35 ± 0.02	0.22 ± 0.02	8.71 ± 1.09
ExtraTreesClassifier23t30d, octets	0.9000	0.9905	0.65 ± 0.03	0.93 ± 0.01	0.35 ± 0.03	0.98 ± 0.00	0.91 ± 0.01	0.45 ± 0.03	0.58 ± 0.02	0.33 ± 0.03	0.44 ± 0.03	0.29 ± 0.03	10.14 ± 1.53
ExtraTreesClassifier23t30d, bits	0.9000	0.9958	0.68 ± 0.05	0.91 ± 0.02	0.52 ± 0.05	0.95 ± 0.01	0.88 ± 0.01	0.59 ± 0.05	0.59 ± 0.03	0.48 ± 0.04	0.53 ± 0.04	0.42 ± 0.05	6.24 ± 1.33
ExtraTreesClassifier23t, raw	0.9000	0.9984	0.79 ± 0.01	0.82 ± 0.01	0.65 ± 0.04	0.90 ± 0.02	0.81 ± 0.01	0.71 ± 0.02	0.61 ± 0.01	0.55 ± 0.02	0.58 ± 0.01	0.55 ± 0.03	2.81 ± 0.15
ExtraTreesClassifier23t, octets	0.9000	0.9971	0.73 ± 0.03	0.88 ± 0.01	0.57 ± 0.06	0.94 ± 0.01	0.86 ± 0.01	0.64 ± 0.05	0.61 ± 0.02	0.51 ± 0.05	0.56 ± 0.04	0.47 ± 0.05	4.57 ± 0.65
ExtraTreesClassifier23t, bits	0.9000	0.9967	0.70 ± 0.05	0.90 ± 0.01	0.57 ± 0.05	0.94 ± 0.01	0.87 ± 0.01	0.63 ± 0.05	0.60 ± 0.03	0.51 ± 0.05	0.55 ± 0.04	0.46 ± 0.05	5.25 ± 1.16
ExtraTreesClassifier, raw	0.9000	0.9984	0.79 ± 0.01	0.82 ± 0.01	0.65 ± 0.05	0.90 ± 0.02	0.81 ± 0.00	0.71 ± 0.02	0.61 ± 0.01	0.55 ± 0.03	0.58 ± 0.01	0.55 ± 0.03	2.83 ± 0.15
ExtraTreesClassifier, octets	0.9000	0.9968	0.71 ± 0.04	0.89 ± 0.01	0.56 ± 0.06	0.94 ± 0.01	0.86 ± 0.01	0.63 ± 0.05	0.60 ± 0.03	0.50 ± 0.05	0.55 ± 0.04	0.46 ± 0.06	5.00 ± 0.96
ExtraTreesClassifier, bits	0.9000	0.9966	0.70 ± 0.06	0.90 ± 0.02	0.57 ± 0.05	0.94 ± 0.01	0.87 ± 0.01	0.63 ± 0.05	0.60 ± 0.04	0.51 ± 0.04	0.55 ± 0.04	0.46 ± 0.05	5.46 ± 1.26

Figure 5.40: Performance metrics for ExtraTreesClassifier for 90% resulting traffic coverage across all 5 folds.

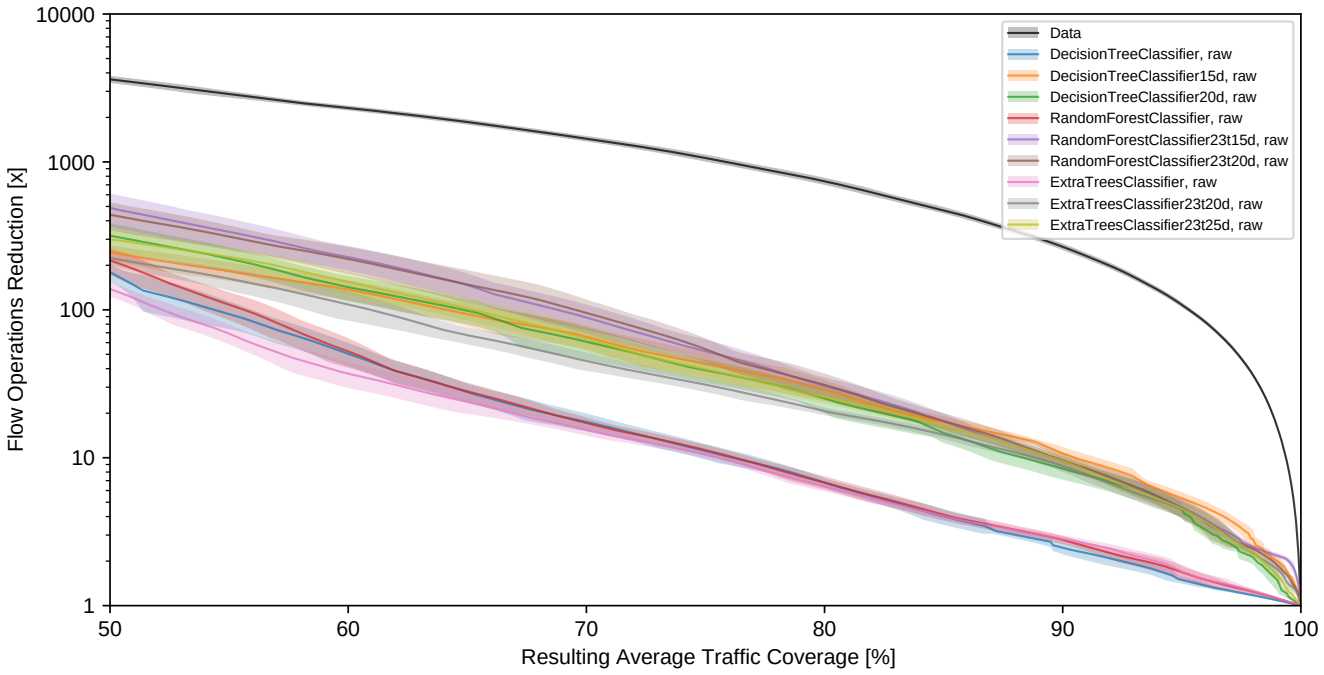

Figure 5.41: Flow operations reduction with standard deviation (shaded line) for *raw* input data.

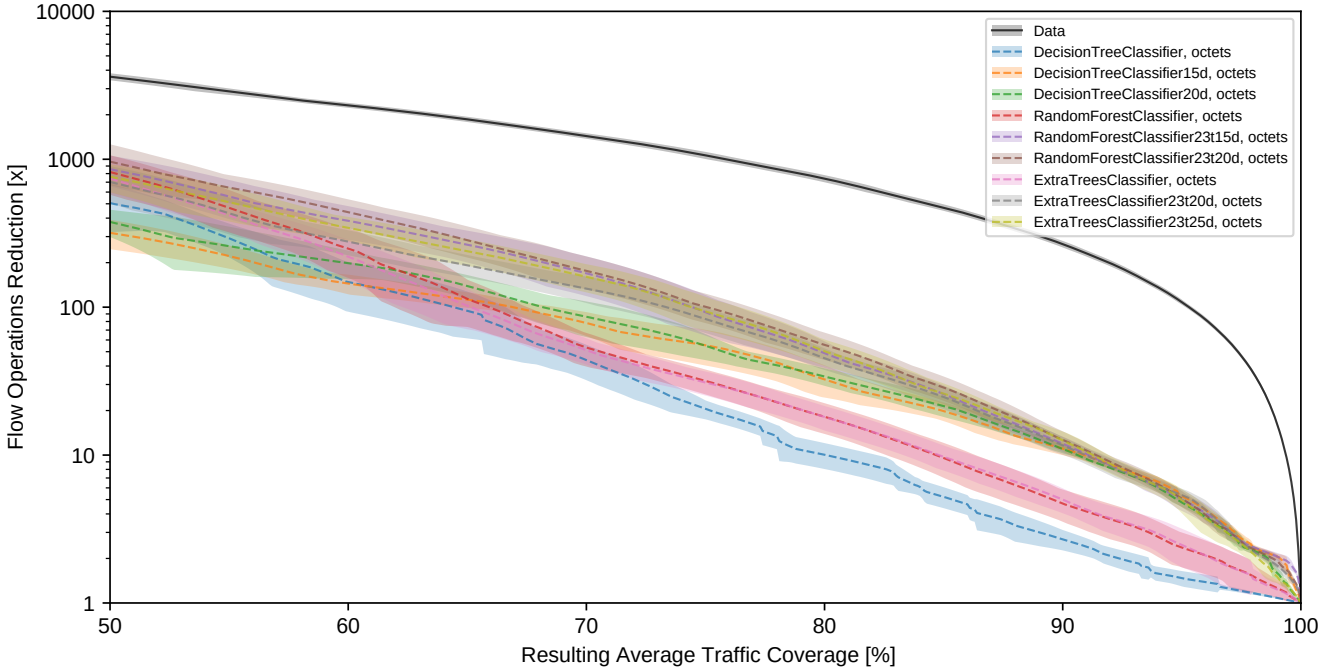

Figure 5.42: Flow operations reduction with standard deviation (shaded line) for *octets* input data.

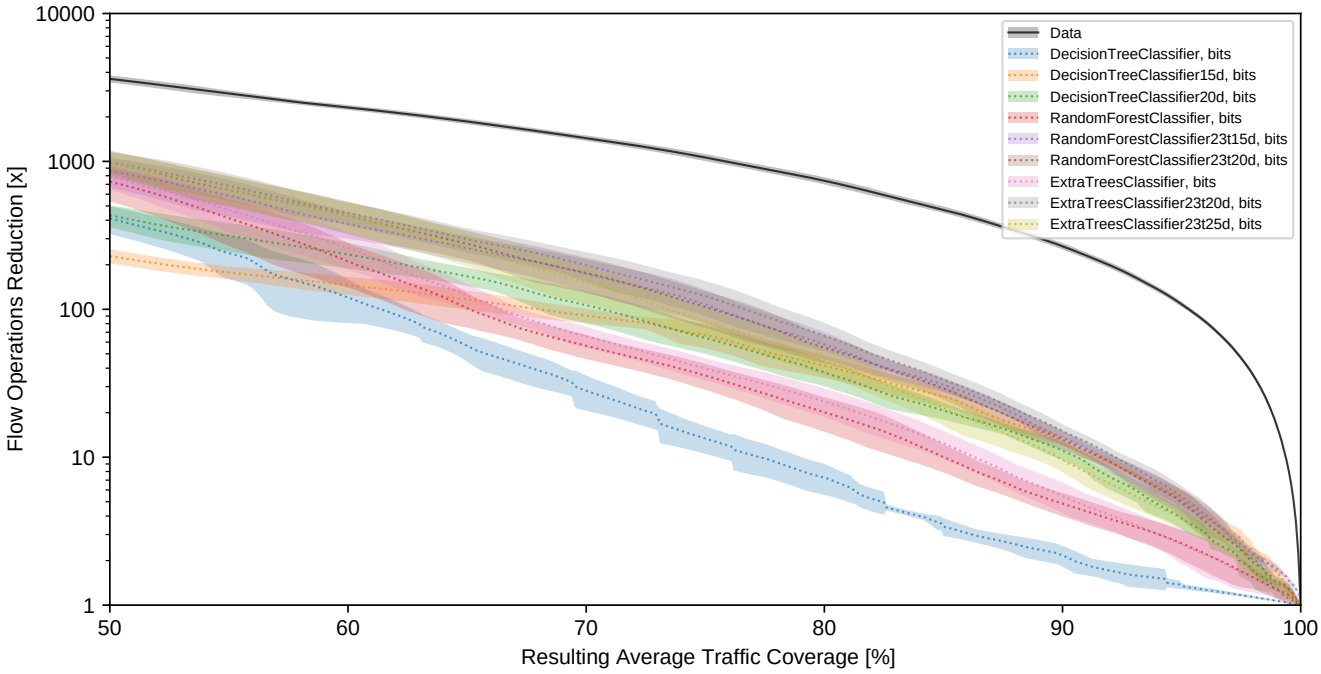

Figure 5.43: Flow operations reduction with standard deviation (shaded line) for *bits* input data.

on the dataset. To manage memory consumption and control tree size and complexity, it is important to fine-tune the model parameters appropriately. In contrast, depth-limited models exhibit a significantly lower number of leaves. For instance, random forests composed of trees with a depth limit of 15 — which were found to be highly effective for reducing flow table entries, as detailed at the beginning of this Section — have less than 10,000 leaves on average. Additionally, it is worth noting that trees using the *bits* input format (binary decision trees) tend to have much more leaves. The number of leaves also scales with traffic coverage: as models are trained to handle higher traffic coverage, they encounter more flows labeled as "elephants" during training, resulting in more complex trees to capture this additional information.

For tree-based models classical ML metrics other than *accuracy* do not correlate with the flow operations reduction as shown in Figures 5.32, 5.35, 5.38, 5.33, 5.36, 5.39, 5.34, 5.37, and 5.40. One can go even further and state that metrics such as *FScore* or *MK* are inversely correlated. Similarly as in the non-tree-based models *TNR* is an exception. The reason behind this is the same - after the normalization, the objective is to minimize the number of flow operations which is a direct effect of the reduction of mice flows missclassified as elephants. This translated to minimizing false positives and therefore maximizing the *TNR*.

As shown in Figure 5.44 *raw* input data representation has the worst performance across all models and resulting traffic coverages. On the other hand, using different data representations (*bits*, *octets*) for ensemble models (*RandomForestClassifier*, *ExtraTreesClassifier*) offers significant improvements. Ensemble models strongly outperform the single tree model, especially with *bits* or *octets* input data representations. However, it is the *bits* format that performs the best in connection with the *ExtraTreesClassifier*.

As for the constrained ensemble models first of all, what is surprising is the fact that decreasing the number of trees from 100 (default) to 23 does not result in significant performance losses - the decrease is very slight. This observation allowed the formulation of a thesis that **the relationship between the en-**

semble size and model effectiveness is not linear. Going even further, what can be observed is the fact that the constraints on the tree depth improved the performance. This is observed across all models - *DecisionTreeClassifier*, *RandomForestClassifier*, and *ExtraTreesClassifier*. At first, it may be counterintuitive. However, after giving it some thought it is not. Limiting tree depth plays a crucial role in preventing models from overfitting to the training data by capturing noise or peculiarities that do not generalize well to new, unseen instances. This improved generalization highlights the significance of employing regularization techniques, which are essential for controlling model complexity, even in ensemble methods that are typically robust, such as random forests. The optimal tree depth, however, is not a one-size-fits-all parameter. It varies depending on several factors, including the format of the input data and the specific classifier being used. For example, models trained on more detailed or high-dimensional data, such as bit-level representations, may require shallower trees to avoid overfitting, while other data formats like octets might benefit from deeper trees to capture more complex patterns. Fine-tuning tree depth is thus a balancing act: while too much depth can lead to large, over-complex trees prone to overfitting, too little depth can limit the model's ability to learn important relationships in the data. Finding the right depth is key to maximizing performance while maintaining computational efficiency.

The best performance of the *DecisionTreeClassifier* was achieved using the *bits* representation and a tree depth of 15. This combination resulted in a flow operation reduction of 41.59 and an *accuracy* of 0.976 for 80% resulting traffic coverage. For the *octets* format, the best performance was observed with a tree depth of 20, which provided a flow operation reduction by a factor of 34 and an *accuracy* of 0.971. As for the *RandomForestClassifier* the optimal depth for the *bits* format is also 15. This combination provided a 57.97 reduction in flow operations and an *accuracy* of 0.981 for the 80% resulting traffic coverage. Similarly as for the *DecisionTreeClassifier* for *octets* the optimal tree depth is 20. Such a combination yielded 55.63 flow operations reduction and an *accuracy* of 0.981. *ExtraTreesClassifier* with *bits* input data performance was the best at a depth of 20 providing 66.35 flow operations reduction, and an *accuracy* of 0.984. For the *octets* input format, the optimal tree depth is 25, resulting in a flow operation reduction of 50.31, and an *accuracy* of 0.979.

The results show that the *bits* input format outperforms the *octets* format across all tree-based classifiers, delivering greater reductions in flow operations along with higher accuracy. Additionally, the optimal tree depth for the *bits* format is consistently shallower than for the *octets* format, indicating that the *bits* representation enables more efficient feature extraction at lower depths. This suggests that the *bits* format not only enhances classifier performance but also improves computational efficiency by requiring less complex tree structures to achieve superior results. Additionally, trees of depth 15, which offer optimal performance for the *bits* data format, contain fewer than 10,000 leaves, while trees of depth 20, yielding the best performance for the *octets* format, have fewer than 20,000 leaves, as shown in Figure 5.46. These findings highlight an interesting trade-off between tree depth and the number of input features. With fewer input features, such as in the *bits* format, shallower trees are sufficient to achieve optimal results. In contrast, the *octets* format requires deeper trees to maintain comparable performance. This balance between tree complexity and input feature representation has important implications for designing and implementing models, especially in

resource-constrained environments like networking hardware, where memory and computational efficiency are critical considerations.

	Resulting Average Traffic Coverage		
	70%	80%	90%
Data, raw	1437.25 ± 62.11	740.03 ± 41.45	266.91 ± 13.66
DecisionTreeClassifier10d, raw	51.43 ± 6.33	19.13 ± 0.81	7.81 ± 0.60
DecisionTreeClassifier10d, octets	54.14 ± 13.58	19.91 ± 3.50	8.16 ± 0.71
DecisionTreeClassifier10d, bits	42.69 ± 2.66	18.75 ± 2.59	8.12 ± 0.80
DecisionTreeClassifier15d, raw	65.85 ± 12.07	28.83 ± 4.68	10.67 ± 1.16
DecisionTreeClassifier15d, octets	78.21 ± 14.41	32.48 ± 8.01	11.07 ± 1.09
DecisionTreeClassifier15d, bits	90.69 ± 10.21	41.59 ± 7.36	12.63 ± 0.67
DecisionTreeClassifier20d, raw	60.84 ± 15.06	25.14 ± 5.10	8.45 ± 1.34
DecisionTreeClassifier20d, octets	86.04 ± 23.27	34.00 ± 4.61	11.00 ± 0.71
DecisionTreeClassifier20d, bits	106.59 ± 26.27	37.42 ± 7.29	11.09 ± 1.56
DecisionTreeClassifier25d, raw	43.10 ± 7.01	15.17 ± 2.96	6.30 ± 1.01
DecisionTreeClassifier25d, octets	74.63 ± 20.33	27.49 ± 5.51	7.83 ± 0.91
DecisionTreeClassifier25d, bits	66.96 ± 17.72	19.44 ± 4.51	5.18 ± 1.02
DecisionTreeClassifier30d, raw	28.05 ± 5.31	10.73 ± 1.44	4.14 ± 0.44
DecisionTreeClassifier30d, octets	57.04 ± 14.60	17.65 ± 3.06	5.93 ± 0.84
DecisionTreeClassifier30d, bits	35.87 ± 6.75	10.93 ± 2.30	2.67 ± 0.47
DecisionTreeClassifier, raw	17.48 ± 2.43	6.77 ± 0.49	2.46 ± 0.23
DecisionTreeClassifier, octets	43.91 ± 11.83	10.05 ± 2.00	2.69 ± 0.42
DecisionTreeClassifier, bits	28.12 ± 7.54	7.29 ± 1.74	2.16 ± 0.35
RandomForestClassifier23t10d, raw	53.99 ± 8.34	17.10 ± 2.69	7.38 ± 1.22
RandomForestClassifier23t10d, octets	92.33 ± 28.84	28.11 ± 4.56	8.28 ± 1.10
RandomForestClassifier23t10d, bits	97.53 ± 24.41	29.35 ± 5.25	8.27 ± 1.23
RandomForestClassifier23t15d, raw	88.74 ± 19.73	30.57 ± 4.41	9.45 ± 0.76
RandomForestClassifier23t15d, octets	169.57 ± 50.79	48.81 ± 9.98	11.92 ± 1.42
RandomForestClassifier23t15d, bits	174.02 ± 43.03	57.97 ± 12.08	13.62 ± 1.58
RandomForestClassifier23t20d, raw	95.17 ± 22.40	31.11 ± 6.00	9.57 ± 1.30
RandomForestClassifier23t20d, octets	176.16 ± 44.42	55.63 ± 11.66	12.69 ± 1.80
RandomForestClassifier23t20d, bits	176.06 ± 47.56	55.04 ± 11.81	13.14 ± 1.93
RandomForestClassifier23t25d, raw	64.69 ± 17.17	21.47 ± 4.01	7.84 ± 1.37
RandomForestClassifier23t25d, octets	143.97 ± 32.00	44.01 ± 10.33	10.64 ± 1.64
RandomForestClassifier23t25d, bits	126.13 ± 42.67	34.14 ± 6.72	7.99 ± 1.54
RandomForestClassifier23t30d, raw	40.38 ± 10.95	15.00 ± 2.40	6.08 ± 1.34
RandomForestClassifier23t30d, octets	102.08 ± 27.70	29.18 ± 6.14	7.99 ± 1.20
RandomForestClassifier23t30d, bits	69.26 ± 14.03	23.75 ± 6.11	5.39 ± 1.10
RandomForestClassifier23t, raw	17.59 ± 2.15	6.83 ± 0.66	2.79 ± 0.21
RandomForestClassifier23t, octets	52.85 ± 16.78	17.01 ± 3.71	4.43 ± 0.89
RandomForestClassifier23t, bits	56.13 ± 12.68	17.89 ± 4.34	4.58 ± 1.00
RandomForestClassifier, raw	17.16 ± 1.55	6.80 ± 0.70	2.78 ± 0.18
RandomForestClassifier, octets	53.32 ± 13.67	18.16 ± 4.27	4.69 ± 1.15
RandomForestClassifier, bits	56.61 ± 10.56	20.14 ± 5.30	4.86 ± 0.89
ExtraTreesClassifier23t10d, raw	16.87 ± 2.95	8.79 ± 0.58	4.82 ± 0.37
ExtraTreesClassifier23t10d, octets	29.53 ± 6.14	13.87 ± 1.24	5.41 ± 0.35
ExtraTreesClassifier23t10d, bits	97.56 ± 27.87	30.01 ± 5.58	8.34 ± 1.26
ExtraTreesClassifier23t15d, raw	29.03 ± 4.34	13.84 ± 0.64	6.56 ± 0.56
ExtraTreesClassifier23t15d, octets	89.42 ± 21.58	29.65 ± 3.79	9.25 ± 1.07
ExtraTreesClassifier23t15d, bits	192.32 ± 45.08	62.91 ± 13.62	14.80 ± 1.80
ExtraTreesClassifier23t20d, raw	45.12 ± 6.45	20.62 ± 1.21	8.80 ± 0.79
ExtraTreesClassifier23t20d, octets	133.82 ± 27.20	45.16 ± 7.54	11.66 ± 1.61
ExtraTreesClassifier23t20d, bits	198.72 ± 44.75	66.35 ± 14.62	15.03 ± 1.68
ExtraTreesClassifier23t25d, raw	63.92 ± 11.39	25.75 ± 2.55	9.47 ± 0.91
ExtraTreesClassifier23t25d, octets	160.56 ± 34.74	50.31 ± 10.16	12.58 ± 1.86
ExtraTreesClassifier23t25d, bits	151.70 ± 43.06	45.25 ± 9.64	9.57 ± 1.66
ExtraTreesClassifier23t30d, raw	71.00 ± 16.52	26.48 ± 4.62	8.71 ± 1.09
ExtraTreesClassifier23t30d, octets	148.95 ± 35.80	43.85 ± 10.62	10.14 ± 1.53
ExtraTreesClassifier23t30d, bits	87.65 ± 19.52	27.38 ± 6.37	6.24 ± 1.33
ExtraTreesClassifier23t, raw	15.63 ± 1.59	6.60 ± 0.47	2.81 ± 0.15
ExtraTreesClassifier23t, octets	49.23 ± 11.00	16.63 ± 3.76	4.57 ± 0.65
ExtraTreesClassifier23t, bits	62.51 ± 11.21	21.85 ± 5.35	5.25 ± 1.16
ExtraTreesClassifier, raw	15.41 ± 1.30	6.38 ± 0.45	2.83 ± 0.15
ExtraTreesClassifier, octets	51.22 ± 10.33	18.05 ± 3.41	5.00 ± 0.96
ExtraTreesClassifier, bits	65.54 ± 10.56	23.88 ± 5.27	5.46 ± 1.26

Flow Operations Reduction (mean ± std) [x]

Figure 5.44: Comparison of flow table performance metrics of tree-based classifiers for 70%, 80%, and 90% resulting traffic coverages.

	Resulting Average Traffic Coverage			Resulting Average Traffic Coverage		
	70%	80%	90%	70%	80%	90%
DecisionTreeClassifier10d, raw	0.938717	0.963406	0.982208	0.979 ± 0.002	0.945 ± 0.003	0.876 ± 0.009
DecisionTreeClassifier10d, octets	0.926007	0.965605	0.983070	0.979 ± 0.009	0.946 ± 0.008	0.880 ± 0.011
DecisionTreeClassifier10d, bits	0.879623	0.950410	0.980628	0.976 ± 0.001	0.946 ± 0.008	0.881 ± 0.011
DecisionTreeClassifier15d, raw	0.904426	0.954139	0.980594	0.984 ± 0.004	0.964 ± 0.005	0.912 ± 0.010
DecisionTreeClassifier15d, octets	0.902476	0.954721	0.982004	0.987 ± 0.003	0.968 ± 0.007	0.915 ± 0.007
DecisionTreeClassifier15d, bits	0.863361	0.935069	0.981189	0.989 ± 0.001	0.976 ± 0.005	0.927 ± 0.004
DecisionTreeClassifier20d, raw	0.934052	0.966182	0.988366	0.982 ± 0.004	0.960 ± 0.009	0.895 ± 0.014
DecisionTreeClassifier20d, octets	0.918664	0.959644	0.985661	0.987 ± 0.004	0.971 ± 0.004	0.919 ± 0.005
DecisionTreeClassifier20d, bits	0.904890	0.959522	0.987530	0.990 ± 0.003	0.973 ± 0.007	0.920 ± 0.010
DecisionTreeClassifier25d, raw	0.956200	0.982318	0.993008	0.975 ± 0.004	0.936 ± 0.012	0.873 ± 0.015
DecisionTreeClassifier25d, octets	0.939789	0.974064	0.991724	0.985 ± 0.004	0.963 ± 0.008	0.894 ± 0.010
DecisionTreeClassifier25d, bits	0.953722	0.983652	0.995842	0.983 ± 0.004	0.946 ± 0.012	0.856 ± 0.018
DecisionTreeClassifier30d, raw	0.974543	0.989488	0.996485	0.961 ± 0.006	0.912 ± 0.009	0.841 ± 0.003
DecisionTreeClassifier30d, octets	0.959465	0.984394	0.994703	0.981 ± 0.004	0.943 ± 0.008	0.870 ± 0.013
DecisionTreeClassifier30d, bits	0.975792	0.992100	0.998514	0.967 ± 0.006	0.907 ± 0.017	0.796 ± 0.015
DecisionTreeClassifier, raw	0.987732	0.995179	0.998701	0.935 ± 0.008	0.868 ± 0.005	0.788 ± 0.008
DecisionTreeClassifier, octets	0.973545	0.992928	0.998534	0.972 ± 0.009	0.901 ± 0.016	0.798 ± 0.014
DecisionTreeClassifier, bits	0.981293	0.995045	0.998996	0.957 ± 0.015	0.873 ± 0.021	0.763 ± 0.017
RandomForestClassifier23t10d, raw	0.942821	0.963951	0.981345	0.978 ± 0.004	0.937 ± 0.007	0.869 ± 0.021
RandomForestClassifier23t10d, octets	0.942012	0.963652	0.982628	0.986 ± 0.005	0.959 ± 0.009	0.884 ± 0.015
RandomForestClassifier23t10d, bits	0.938504	0.962376	0.981373	0.987 ± 0.004	0.959 ± 0.007	0.884 ± 0.014
RandomForestClassifier23t15d, raw	0.936612	0.961848	0.981748	0.987 ± 0.003	0.965 ± 0.008	0.900 ± 0.007
RandomForestClassifier23t15d, octets	0.915927	0.959836	0.981989	0.993 ± 0.003	0.978 ± 0.005	0.921 ± 0.009
RandomForestClassifier23t15d, bits	0.897753	0.954997	0.980546	0.994 ± 0.003	0.981 ± 0.004	0.931 ± 0.007
RandomForestClassifier23t20d, raw	0.945102	0.967563	0.985553	0.987 ± 0.003	0.964 ± 0.007	0.903 ± 0.011
RandomForestClassifier23t20d, octets	0.930177	0.964009	0.986237	0.993 ± 0.002	0.981 ± 0.004	0.928 ± 0.010
RandomForestClassifier23t20d, bits	0.923355	0.964473	0.987590	0.993 ± 0.003	0.981 ± 0.006	0.931 ± 0.011
RandomForestClassifier23t25d, raw	0.961795	0.979131	0.990839	0.981 ± 0.005	0.951 ± 0.009	0.888 ± 0.015
RandomForestClassifier23t25d, octets	0.951488	0.976380	0.990810	0.991 ± 0.002	0.974 ± 0.007	0.918 ± 0.011
RandomForestClassifier23t25d, bits	0.960001	0.982753	0.994073	0.988 ± 0.005	0.965 ± 0.007	0.899 ± 0.015
RandomForestClassifier23t30d, raw	0.976778	0.987329	0.994249	0.968 ± 0.008	0.932 ± 0.009	0.869 ± 0.019
RandomForestClassifier23t30d, octets	0.967917	0.984921	0.994001	0.985 ± 0.005	0.961 ± 0.008	0.900 ± 0.011
RandomForestClassifier23t30d, bits	0.980117	0.990245	0.996706	0.976 ± 0.005	0.945 ± 0.012	0.872 ± 0.012
RandomForestClassifier23t, raw	0.990830	0.995933	0.998543	0.929 ± 0.005	0.871 ± 0.006	0.811 ± 0.008
RandomForestClassifier23t, octets	0.982690	0.992423	0.997462	0.970 ± 0.008	0.930 ± 0.012	0.859 ± 0.007
RandomForestClassifier23t, bits	0.984306	0.992744	0.997406	0.969 ± 0.007	0.929 ± 0.014	0.862 ± 0.010
RandomForestClassifier, raw	0.990804	0.995921	0.998536	0.929 ± 0.004	0.870 ± 0.007	0.812 ± 0.006
RandomForestClassifier, octets	0.983174	0.992212	0.997300	0.970 ± 0.007	0.932 ± 0.012	0.864 ± 0.010
RandomForestClassifier, bits	0.985018	0.992467	0.997235	0.968 ± 0.006	0.933 ± 0.014	0.868 ± 0.008
ExtraTreesClassifier23t10d, raw	0.954191	0.965343	0.986915	0.928 ± 0.007	0.882 ± 0.007	0.807 ± 0.011
ExtraTreesClassifier23t10d, octets	0.959270	0.967468	0.983706	0.958 ± 0.006	0.912 ± 0.006	0.826 ± 0.009
ExtraTreesClassifier23t10d, bits	0.938205	0.961496	0.980578	0.987 ± 0.005	0.961 ± 0.011	0.884 ± 0.015
ExtraTreesClassifier23t15d, raw	0.948081	0.962060	0.978375	0.961 ± 0.005	0.925 ± 0.003	0.855 ± 0.011
ExtraTreesClassifier23t15d, octets	0.939401	0.961725	0.978494	0.986 ± 0.005	0.962 ± 0.007	0.895 ± 0.011
ExtraTreesClassifier23t15d, bits	0.884614	0.950318	0.978336	0.995 ± 0.002	0.983 ± 0.004	0.936 ± 0.007
ExtraTreesClassifier23t20d, raw	0.942642	0.960900	0.978893	0.975 ± 0.003	0.951 ± 0.004	0.892 ± 0.009
ExtraTreesClassifier23t20d, octets	0.919941	0.959434	0.980224	0.992 ± 0.002	0.976 ± 0.005	0.918 ± 0.011
ExtraTreesClassifier23t20d, bits	0.900785	0.955168	0.984749	0.995 ± 0.002	0.984 ± 0.004	0.941 ± 0.007
ExtraTreesClassifier23t25d, raw	0.938848	0.962298	0.982654	0.982 ± 0.003	0.960 ± 0.005	0.901 ± 0.009
ExtraTreesClassifier23t25d, octets	0.912338	0.962464	0.984600	0.993 ± 0.002	0.979 ± 0.005	0.926 ± 0.012
ExtraTreesClassifier23t25d, bits	0.943729	0.974526	0.992244	0.992 ± 0.003	0.975 ± 0.006	0.913 ± 0.013
ExtraTreesClassifier23t30d, raw	0.943346	0.969021	0.988449	0.983 ± 0.004	0.960 ± 0.008	0.895 ± 0.013
ExtraTreesClassifier23t30d, octets	0.934420	0.972572	0.990467	0.992 ± 0.003	0.975 ± 0.008	0.914 ± 0.012
ExtraTreesClassifier23t30d, bits	0.970699	0.987010	0.995764	0.983 ± 0.004	0.956 ± 0.011	0.882 ± 0.015
ExtraTreesClassifier23t, raw	0.990949	0.995722	0.998446	0.923 ± 0.005	0.866 ± 0.005	0.812 ± 0.005
ExtraTreesClassifier23t, octets	0.980886	0.991541	0.997066	0.971 ± 0.005	0.931 ± 0.012	0.859 ± 0.007
ExtraTreesClassifier23t, bits	0.979290	0.990283	0.996714	0.976 ± 0.005	0.943 ± 0.014	0.869 ± 0.011
ExtraTreesClassifier, raw	0.991160	0.995892	0.998432	0.922 ± 0.004	0.864 ± 0.005	0.814 ± 0.004
ExtraTreesClassifier, octets	0.981081	0.991246	0.996793	0.971 ± 0.006	0.935 ± 0.009	0.864 ± 0.010
ExtraTreesClassifier, bits	0.979500	0.989993	0.996603	0.976 ± 0.004	0.946 ± 0.012	0.872 ± 0.013
	Training Traffic Coverage (mean)			Accuracy (mean ± std)		

Figure 5.45: Comparison of training coverage and accuracy metric of tree-based classifiers for 70%, 80%, and 90% resulting traffic coverages.

	Resulting Average Traffic Coverage		
	70%	80%	90%
DecisionTreeClassifier10d, raw	629	701	778
DecisionTreeClassifier10d, octets	691	740	793
DecisionTreeClassifier10d, bits	840	923	951
DecisionTreeClassifier15d, raw	3842	5317	6753
DecisionTreeClassifier15d, octets	4011	5516	7448
DecisionTreeClassifier15d, bits	5687	8437	11064
DecisionTreeClassifier20d, raw	14981	21099	32689
DecisionTreeClassifier20d, octets	12227	19153	32854
DecisionTreeClassifier20d, bits	19002	36655	62520
DecisionTreeClassifier25d, raw	37373	71538	103976
DecisionTreeClassifier25d, octets	29079	53025	99370
DecisionTreeClassifier25d, bits	61586	140578	288601
DecisionTreeClassifier30d, raw	88536	169106	250383
DecisionTreeClassifier30d, octets	57044	125185	205293
DecisionTreeClassifier30d, bits	135395	337695	639053
DecisionTreeClassifier, raw	279838	534392	866670
DecisionTreeClassifier, octets	123245	379076	745094
DecisionTreeClassifier, bits	183169	480030	891895
RandomForestClassifier23t10d, raw	591	633	664
RandomForestClassifier23t10d, octets	679	704	749
RandomForestClassifier23t10d, bits	849	884	921
RandomForestClassifier23t15d, raw	3562	4203	5054
RandomForestClassifier23t15d, octets	4134	5248	6620
RandomForestClassifier23t15d, bits	6149	8598	11104
RandomForestClassifier23t20d, raw	11231	14661	20320
RandomForestClassifier23t20d, octets	13129	18643	28611
RandomForestClassifier23t20d, bits	22867	37169	63076
RandomForestClassifier23t25d, raw	27379	39567	58633
RandomForestClassifier23t25d, octets	32107	52042	84496
RandomForestClassifier23t25d, bits	72797	136023	232774
RandomForestClassifier23t30d, raw	62926	91009	136096
RandomForestClassifier23t30d, octets	65677	114474	183974
RandomForestClassifier23t30d, bits	174464	298466	495005
RandomForestClassifier23t, raw	262406	428675	608949
RandomForestClassifier23t, octets	163230	310361	515874
RandomForestClassifier23t, bits	231445	406450	635348
RandomForestClassifier, raw	262137	428595	607652
RandomForestClassifier, octets	166252	306594	506318
RandomForestClassifier, bits	239774	397787	621299
ExtraTreesClassifier23t10d, raw	551	568	631
ExtraTreesClassifier23t10d, octets	666	685	718
ExtraTreesClassifier23t10d, bits	876	910	939
ExtraTreesClassifier23t15d, raw	3524	3826	4473
ExtraTreesClassifier23t15d, octets	4749	5476	6507
ExtraTreesClassifier23t15d, bits	6668	9245	11941
ExtraTreesClassifier23t20d, raw	11205	13129	17152
ExtraTreesClassifier23t20d, octets	14411	20171	28309
ExtraTreesClassifier23t20d, bits	22883	38670	67669
ExtraTreesClassifier23t25d, raw	25170	32773	50445
ExtraTreesClassifier23t25d, octets	29157	51306	85168
ExtraTreesClassifier23t25d, bits	70643	129759	259113
ExtraTreesClassifier23t30d, raw	49842	75887	136313
ExtraTreesClassifier23t30d, octets	61609	117647	220025
ExtraTreesClassifier23t30d, bits	174036	338701	610277
ExtraTreesClassifier23t, raw	660103	942907	1372929
ExtraTreesClassifier23t, octets	341917	617830	989248
ExtraTreesClassifier23t, bits	265950	492555	850325
ExtraTreesClassifier, raw	668356	959851	1367898
ExtraTreesClassifier, octets	347045	604586	964194
ExtraTreesClassifier, bits	268123	481003	841474

Average Number of Tree Leaves (mean)

Figure 5.46: Average number of leaves in each tree.

5.4 Conclusion

As shown in Section 5 **it is possible to accurately classify elephant flows right at the start using only information stored in the 5-tuple.** For that purpose, various normalization methods, input data representations, problem formulations (regression/classification), and models were designed. The best-performing models are summarized in Table 5.11. **Across all tested models and problem formulation, *bits* input data representation showed superiority.** However, having 104 input features compared to 13 input features (*octets*) provides additional memory usage and computation time.

To classify a flow as an elephant in a regression problem two types of ML models were tested: **simple neural networks and TabNet**. For those models, it was necessary to balance the dataset appropriately and normalize the labels before the training phase to obtain satisfactory results. Additionally, for the TabNet model more in-depth analysis of the feature significance and entropy was examined. Such detection of essential features makes it possible to further reduce the model's complexity by removing unnecessary input data. This simplification results in faster training and evaluation, which is especially beneficial for accelerating elephant flow classification and minimizing added latency in the network. Achieving such operational efficiency is crucial for preserving optimal network performance and responsiveness. From all variations of the simple neural networks, their balancing ratios, normalization methods, and hyperparameter variations the best performing was the FourLayerNeuralNetworkRegression which achieved a 14.7 reduction of flow operations for 80% traffic coverage. TabNet on the other hand with the most optimal combination of balancing ratios, normalization methods, and hyperparameter variations achieved a 20.14 reduction of flow operations for 80% traffic coverage.

The classification problem was also approached with two types of ML models: **non-tree-based, and tree-based**. Apart from the flow classification metrics defined at the beginning of this Chapter, traditional ML metrics such as *accuracy* were evaluated. **As expected, traditional metrics fall short in reflecting application-specific performance indicators such as flow operations reduction**. To make traditional metrics such as accuracy serve as a proxy for these application-specific outcomes, they must be compared under the same resulting traffic coverage. However, even after this normalization, the correlation between accuracy and flow-specific metrics remains non-linear. For a model to be considered valuable in traffic engineering and QoS applications, it must achieve accuracy exceeding 90% post-normalization. This threshold ensures the model's reliability in practical scenarios where flow management and optimization are critical. For models evaluated in Section 5.3 instead of dataset balancing, sample weights during the training phase were used. The square root of flow size as a sample weight significantly improved performance compared to using non-weighted training data.

For tree-based models, the results demonstrate that ensemble methods, particularly extremely randomized trees, significantly outperform single-decision tree models. However, this performance boost is dependent on the input data format. Both the bits and octets representations consistently delivered superior results compared to raw data formats across all classifiers, highlighting the importance of data preprocessing in optimizing model performance. Additionally, the results showed that restricting tree depth resulted in better classification performance, likely due to reduced overfitting and improved generalization. The optimal tree depths varied depending on the classifier and input format, typically ranging between 15 and 25 levels. In

addition to that, an interesting trade-off between tree depth and the number of input features was recognized. Models utilizing the *bits* format reached peak performance at shallower depths than those using the *octets* format. This insight is particularly important for model design in resource-constrained environments like networking hardware.

From the non-tree-based models, the best performing was the Histogram-based Gradient Boosting which for 80% resulting traffic coverage, reduces the number of flow operations by a factor of 36.49. When it comes to the tree-based models, the best performing was the constrained Extremely Randomized Tree classifier reducing flow operations by a factor of 66.35 for 80% resulting traffic coverage.

Table 5.11: Best classification models.

Model	**Flow Operations Reduction**	**Library**
FourLayerNeuralNetworkRegression	14.7	*PyTorch*
TabNet	20.14	*PyTorch*
HistGradientBoostingClassifier	36.49	*scikit-learn*
ExtraTreesClassifier	**66.35**	*scikit-learn*

Chapter 6

Elephant-only Traffic Engineering

6.1 Concept

Section 5 concentrated on the specific challenge of elephant flow classification in isolation. However, **this section introduces a broader concept of how a ML model can be integrated into a comprehensive traffic engineering system.** Any of the models discussed in Section 5 could theoretically be applied in such a system, depending on the desired balance between complexity, accuracy, and efficiency. Nevertheless, the results presented here focus on the performance of the TE system utilizing the best-performing model from Section 5 the constrained *ExtraTreesClassifier*. This choice highlights the model's ability to effectively manage traffic flows in real-world scenarios while minimizing computational overhead, making it a practical solution for dynamic and large-scale networks where quick and accurate classification is critical for optimizing performance.

Figure 6.1 illustrates a flowchart depicting the operation of the switch data plane and the SDN controller. This packet processing pipeline can be implemented, for instance, in a P4 switch. Each incoming packet, whether it is part of a new flow or a subsequent packet of an existing flow, is checked against the flow table to locate a matching flow entry. The flow table's primary function is to store complete flow entries for elephant flows, which may include the next hop for the flow's path and other flow-specific QoS attributes. If a corresponding entry is found, the packet is forwarded according to the next hop assigned to that flow.

When there is no flow entry in the flow table, the packet is sent to the SDN controller. In the controller, the flow features are extracted from the packet's header 5-tuple. Then, those features undergo an additional preprocessing step to prepare them for further analysis (for example bits extraction). Next, the preprocessed data is fed into the ML model, which was previously trained offline based on historical traffic observations. During the inference step, depending on the model in use, either a regression or a classification model is applied. If a regression model is used, it predicts a normalized flow size, which is then renormalized. The predicted flow size is compared against a configurable elephant size threshold, which can be adjusted by the network administrator according to current requirements. In contrast, if a classification model is employed, the model directly categorizes the flow as either a mouse or an elephant, bypassing the need for size renormalization. In both scenarios, the final step involves determining whether the flow belongs to the mouse or elephant class based on the output of the selected model. **Flows classified as elephants are added**

to the flow table, along with a dedicated path. This path can either be chosen locally from a predefined set of alternative paths, or the packet may be forwarded to the controller for a load-aware selection based on a global view of the network. On the other hand, flows classified as mice are forwarded according to the shortest paths.

In addition to the *classification* component, the concept also incorporates a FAMTAR mechanism. When a link is approaching congestion, the adjacent router increases the cost of that link to a predefined high value, marking it as congested. This updated link cost is propagated through the network as a standard topology change message in the routing protocol. Upon receiving this update, routers recalculate paths that are likely to avoid the congested link and update their routing tables accordingly. However, this adjustment only applies to new flows, while pre-existing flows continue to follow their original paths, as stored in the FFT. Although congested links still carry flows that were active before the congestion was detected, no new transmissions are initiated on those links. Once the congestion subsides, the link cost gradually returns to its original value. It is important to note that FAMTAR relies on routers detecting congestion based on link load. This component is summarized in Figure 6.2.

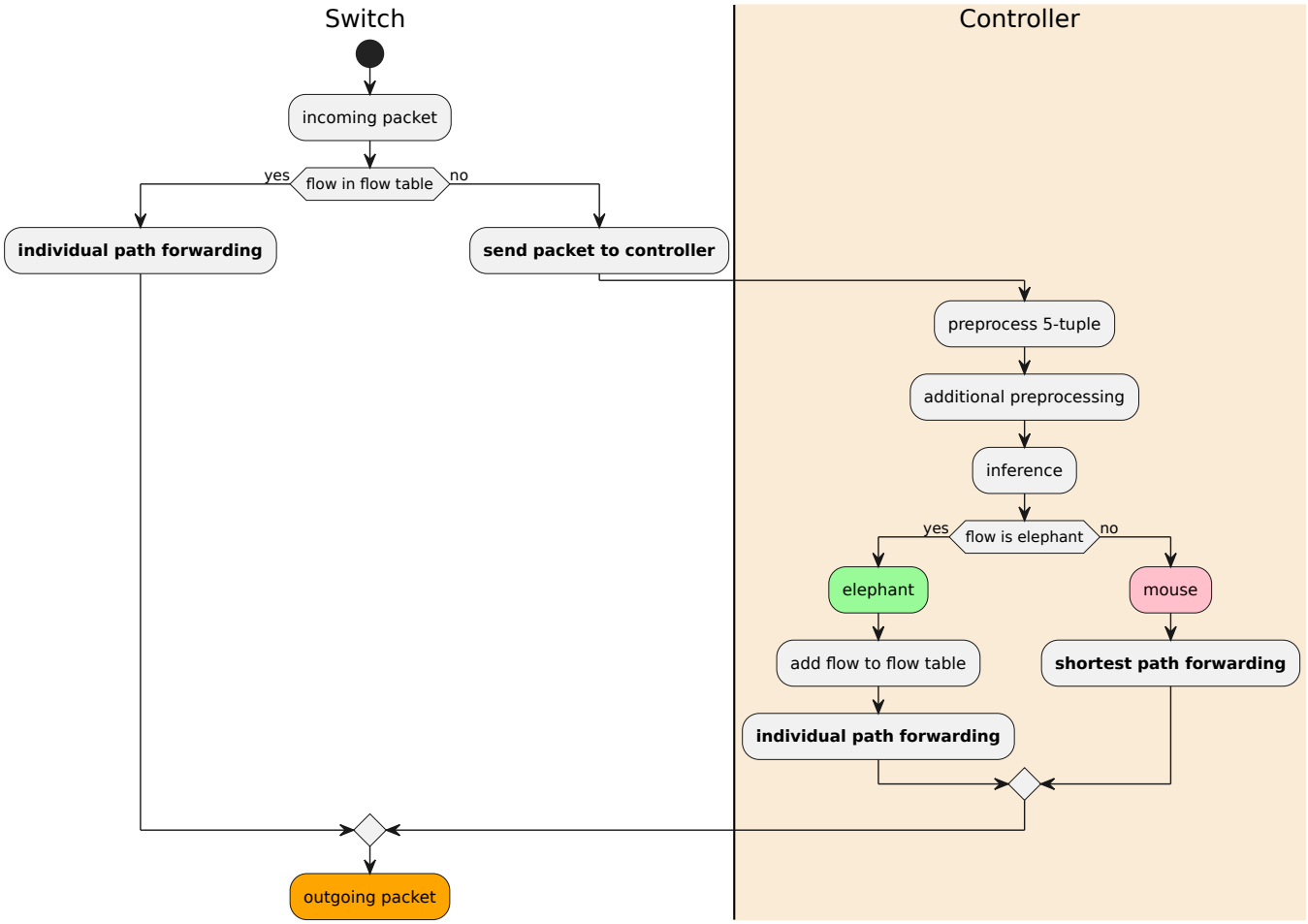

Figure 6.1: Elephant-based TE *classification* component.

A very straightforward limitation of the approach detailed above is that every subsequent packet of a flow must be sent to the controller for classification when no flow table entry exists for mice

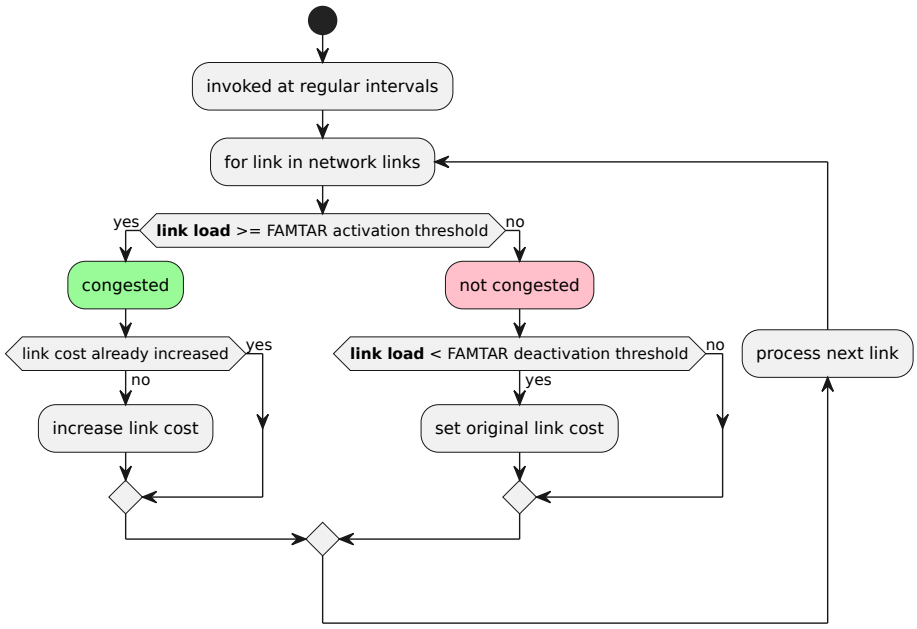

Figure 6.2: Elephant-based TE *FAMTAR* component.

flows, as the switch lacks a caching mechanism to recognize and handle such flows locally. This results in increased controller overhead and added latency, especially for short-lived or mice flows, as they are repeatedly classified by the controller rather than being managed directly by the switch. Consequently, this can lead to inefficient use of network resources and slower packet processing for small flows. A potential solution to this problem is shown in Figure 6.3. When a flow entry is missing in the flow table, the hash of the packet's 5-tuple is checked against a Bloom filter. The Bloom filter functions as a cache for previously observed and already classified mouse flows. If the flow is found in the Bloom filter, it indicates that the flow is a mouse flow, and the packet is forwarded along the shortest paths. Conversely, if the flow is not found in the Bloom filter, it signifies that the incoming packet belongs to a new flow, which requires classification. For flows classified as mice, a hash of the 5-tuple is generated and added to the Bloom filter. This approach prevents subsequent packets of already classified mouse flows from being classified repeatedly. However, since Bloom filters can produce false positive matches, there is a possibility that new flows, which have not yet been classified, might be mistakenly identified as already classified mouse flows. In such cases, these flows will still be forwarded using the shortest paths, following traditional routing methods. While this may slightly reduce the effectiveness of the TE system, it will not disrupt its overall functionality. Additionally, the likelihood of false positives can be precisely managed by adjusting the size of the Bloom filter. To efficiently manage and clean up entries in the Bloom filter, an aging mechanism needs to be implemented. One potential approach involves using a rotating set of Bloom subfilters arranged in a ring structure. For instance, with a 2-second timeout for inactive flows and 10 subfilters, each subfilter would represent a 200-millisecond segment of traffic. When checking for an entry, all subfilters are queried simultaneously, but new entries are added only to the active subfilter at the first position. Every 200 milliseconds, the ring rotates, shifting the subfilters, and the subfilter at the last position is reset to remove stale entries. If an existing flow is

detected, an entry is also added to the current active subfilter to prevent active flows from being prematurely aged. This approach ensures that only inactive flow entries are aged out and cleaned up effectively.

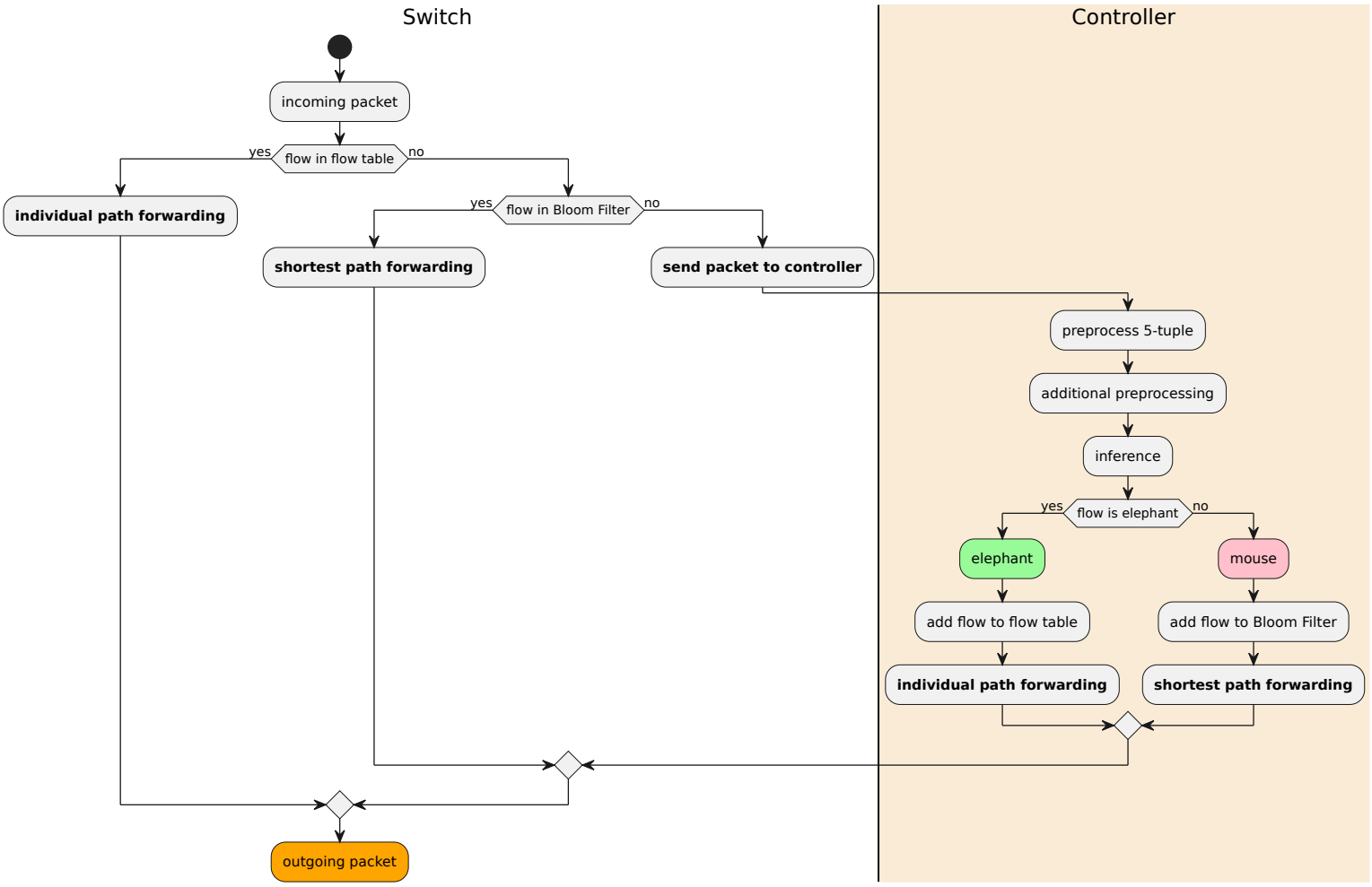

Figure 6.3: Implementation consideration for TE *classification* component including cache mechanism.

6.2 Evaluation

6.2.1 Setup

The concept was evaluated using simulations conducted in a heavily modified version of the Mininet[1] network emulator. A custom-built, Python-based SDN controller that supported only the necessary operations was utilized in the control plane. The controller was responsible for managing network routing and traffic engineering. In the data plane, the Click-based [63] FAMTAR router implementation from [54] was extended by adding components required to communicate with the controller. At the start of each simulation, *veth* Linux tunnels were established between nodes: direct IP layer interfaces for nodes connected in the emulated topology. The direct interfaces were used for IP layer routing and FAMTAR operations.

[1] https://mininet.org/

The performance of the proposed solution was evaluated under various parameters selection. Figure 6.4 presents the network topology used in the experiments. Simulation parameters are summarized in Table 6.1. UDP traffic generators were deployed at each of the 12 nodes, and network traffic was generated based on realistic flow distribution mixtures provided in [53]. These models were derived from a 30-day NetFlow trace recorded at the edge of a campus network, capturing 4 billion flows. Each traffic source generated traffic following a realistic daily pattern. This pattern was based on the average traffic in bits per second that passed through the DE-CIX Internet exchange[2] between midnight CET on January 22, 2019, and midnight CET on January 23, 2019. The traffic envelope is described in depth in [15]. This approach allowed to simulation of a fully realistic load, even though the traffic itself was generated.

Each simulation ran for 5 hours of wall time, where one hour represented a full daily traffic cycle. Additionally, *two different controller operation modes were examined*:

- **NO_PASS** is a classic reactive mode as in OpenFlow, meaning that packets from flows not present in the flow table are sent entirely to the controller. After the controller adds a route for them to the flow tables, it sends the packet back to the switch from which it received it, and only then is the packet forwarded.

- **PASS** is a mode in which packets from flows that are not present in the flow table on the router are still forwarded based on the routing table, while only the packet header is sent to the controller.

The second input parameter was the *traffic generator rate*. Simulations were performed with various levels of network saturation defined as:

- Saturated network: 150 Mbps

- Heavily saturated network: 180 Mbps

- Over-saturated network: 210, 240 Mbps

Table 6.1: Simulation parameters

Paramter	**Value**
FAMTAR activation threshold	0.95
FAMTAR deactivation threshold	0.85
Simulation length [hours]	5
Number of nodes	12
Number of links	18 (bidirectional)
Link bandwidth	100 Mbps
UDP generators rates [Mbps]	150, 180, 210, 240
Operation modes	NO_PASS, PASS

[2]https://www.de-cix.net/

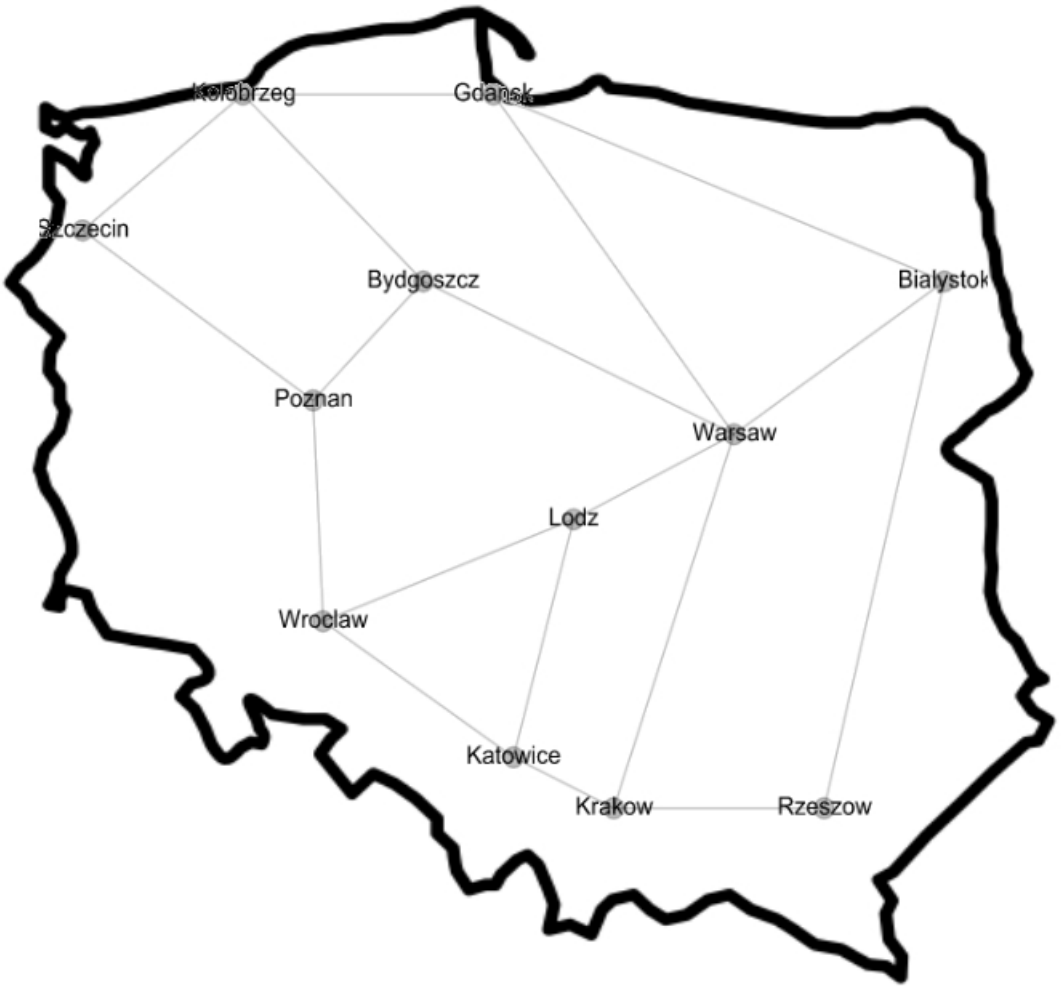

Figure 6.4: Polish network topology.

6.2.2 Classiification inference mechanism

The online inference process proved to be a bottleneck, with a latency of approximately 20 milliseconds per packet. This equates to a processing rate of only 50 packets per second, which is insufficient for high-speed networks where packet rates can reach millions per second. The inherent delay in *scikit-learn's* inference mechanism, while suitable for offline tasks, presents challenges when applied to real-time network environments, limiting its efficiency for on-the-fly packet classification in dynamic traffic engineering scenarios.

Therefore, the classification was shifted to an offline process. All flow classification decisions were pre-calculated and stored before running the simulations. By computing the results offline, it was possible to avoid the latency introduced by real-time inference, ensuring that the traffic engineering simulations could proceed efficiently without a noticeable delay in live classification. This approach allowed for the integration of classification models while maintaining simulation performance. **With these pre-classified flows, the average inference time was reduced to 97.63 nanoseconds per packet, which corresponds to**

a throughput of 10.24 million packets per second. This is more than **204,835** times faster than the online inference time obtained initially with the *scikit-learn*.

However, to achieve this efficiency, it was necessary to modify the training process. The model was retrained using only the first half of the dataset (first 3,258,742 flows corresponding to the 273,5 GB of data transmission as detailed at the beginning of Section 5), ensuring that the predictions generated for the second half of the dataset could be considered independent from the training phase. The second half of the dataset was then used to pre-calculate the classification decisions, which were subsequently applied in the UDP traffic simulations. Additionally, only the second half of the dataset was utilized for generating UDP traffic in the TE simulations, ensuring the integrity and validity of the pre-calculated classification results. This approach not only streamlined the classification process but also ensured that real-time performance was maintained without compromising accuracy or reliability.

6.2.3 Results

Every simulation was repeated 5 times to calculate the standard deviation, ensuring the reliability and consistency of the results. The concept described in Section 6.1 is compared to a system without an elephant classification mechanism, where every flow is added to the FFT. This comparison highlights the effectiveness of the proposed method in distinguishing and managing elephant flows. The results are analyzed in terms of key performance indicators such as link throughput, link delays, percentage of overloaded links, amount of control traffic, controller Central Processing Unit (CPU) usage, number of Shortest Path First (SPF) calculations, and amount of UDP losses. Additionally, the impact of the elephant flow classification on flow table utilization was evaluated (with metrics such as FFT size and add operations), demonstrating how the proposed mechanism reduces memory overhead and processing delays within the network switches. These metrics provide a comprehensive understanding of the benefits and trade-offs of the implemented solution. The figures below can be grouped into 3 categories besides the operation mode and generator rates:

- **link statistics** such as link throughput, link delay, and percentage of overloaded links,
- **controller statistics** such as amount of control traffic, controller CPU usage, and the number of SPF calculations,
- **losses** containing network losses.

For the *NO_PASS* operation mode, results are presented for generator rates of 150, 180, 210, and 240. At a generator rate of 150, Figures 6.5, 6.6, and 6.7 depict the link statistics, while Figures 6.8, 6.9, and 6.10 illustrate control traffic and controller resource usage. UDP losses for this rate are shown in Figure 6.13. Similarly, for a generator rate of 180, Figures 6.23, 6.24, and 6.25 present the link statistics, Figures 6.26, 6.27, and 6.28 display control traffic and controller resource usage, and UDP losses are illustrated in Figure 6.31. At a generator rate of 210, Figures 6.41, 6.42, and 6.43 provide link statistics, Figures 6.44, 6.45, and 6.46 show control traffic and controller resource usage, and UDP losses are depicted in Figure 6.49. Finally, for a generator rate of 240, Figures 6.59, 6.60, and 6.61 illustrate link statistics, Figures 6.62, 6.63, and 6.64 present control traffic and controller resource usage, and UDP losses are displayed in Figure 6.67.

For the *PASS* operation mode, at a generator rate of 150, Figures 6.14, 6.15, and 6.16 depict the link statistics, while Figures 6.17, 6.18, and 6.19 illustrate control traffic and controller resource usage. UDP losses for this rate are shown in Figure 6.22. Similarly, for a generator rate of 180, Figures 6.32, 6.33, and 6.34 present the link statistics, Figures 6.35, 6.36, and 6.37 display control traffic and controller resource usage, and UDP losses are illustrated in Figure 6.40. At a generator rate of 210, Figures 6.50, 6.51, and 6.52 provide link statistics, Figures 6.53, 6.54, and 6.55 show control traffic and controller resource usage, and UDP losses are depicted in Figure 6.58. Finally, for a generator rate of 240, Figures 6.68, 6.69, and 6.70 illustrate link statistics, Figures 6.71, 6.72, and 6.73 present control traffic and controller resource usage, and UDP losses are displayed in Figure 6.76.

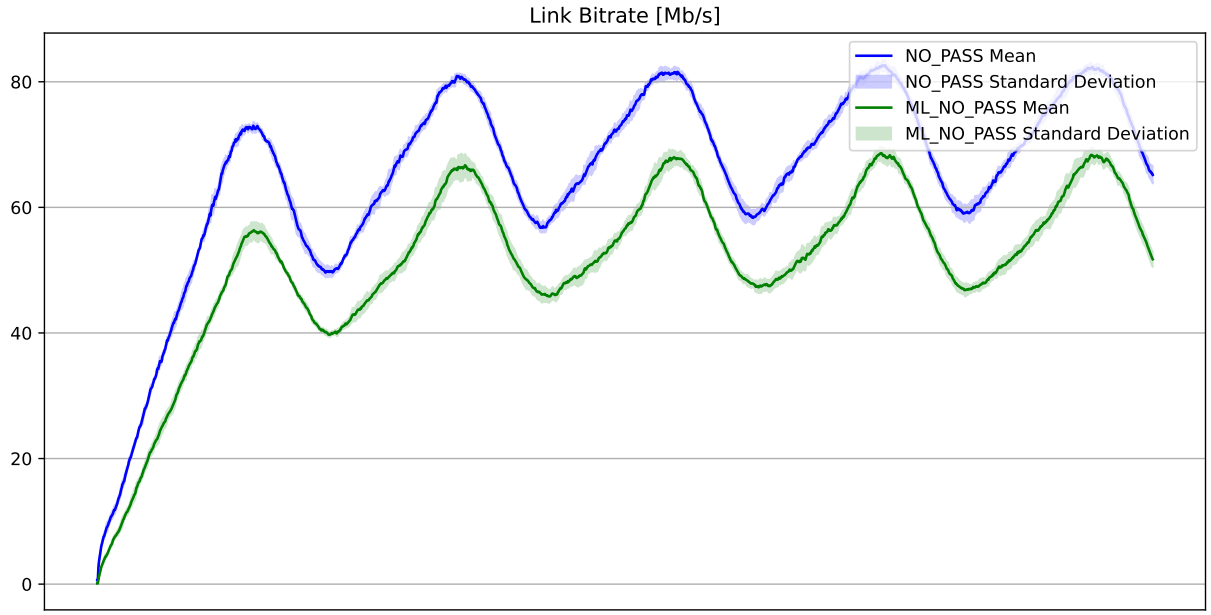

Figure 6.5: Link throughput for *NO_PASS* and *150* generator rate.

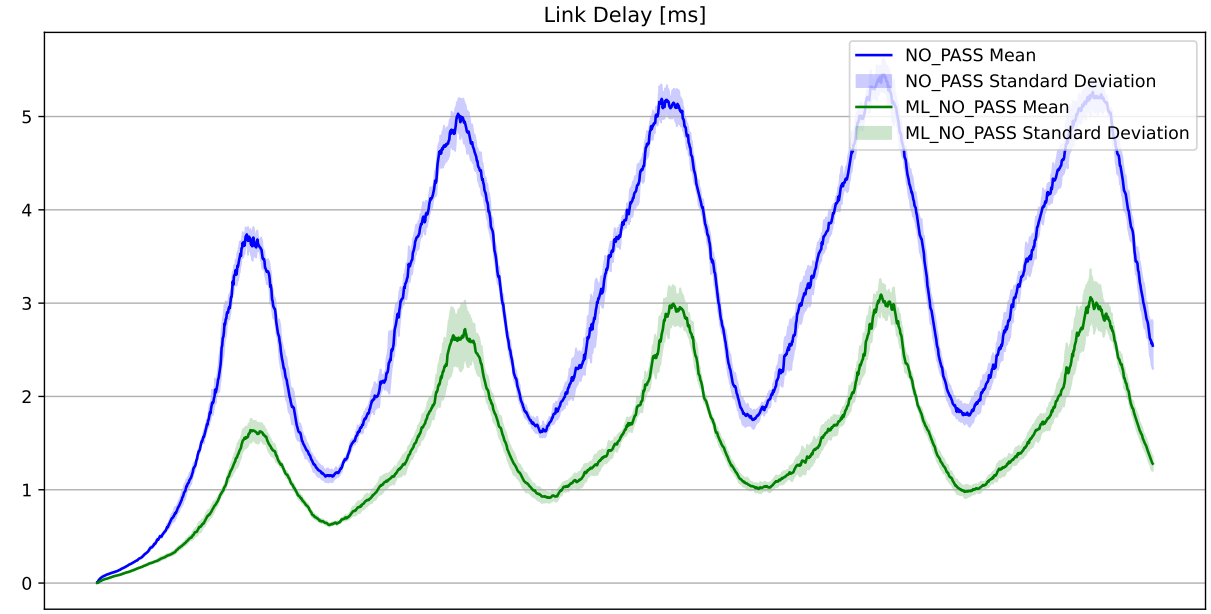

Figure 6.6: Link delay for *NO_PASS* and *150* generator rate.

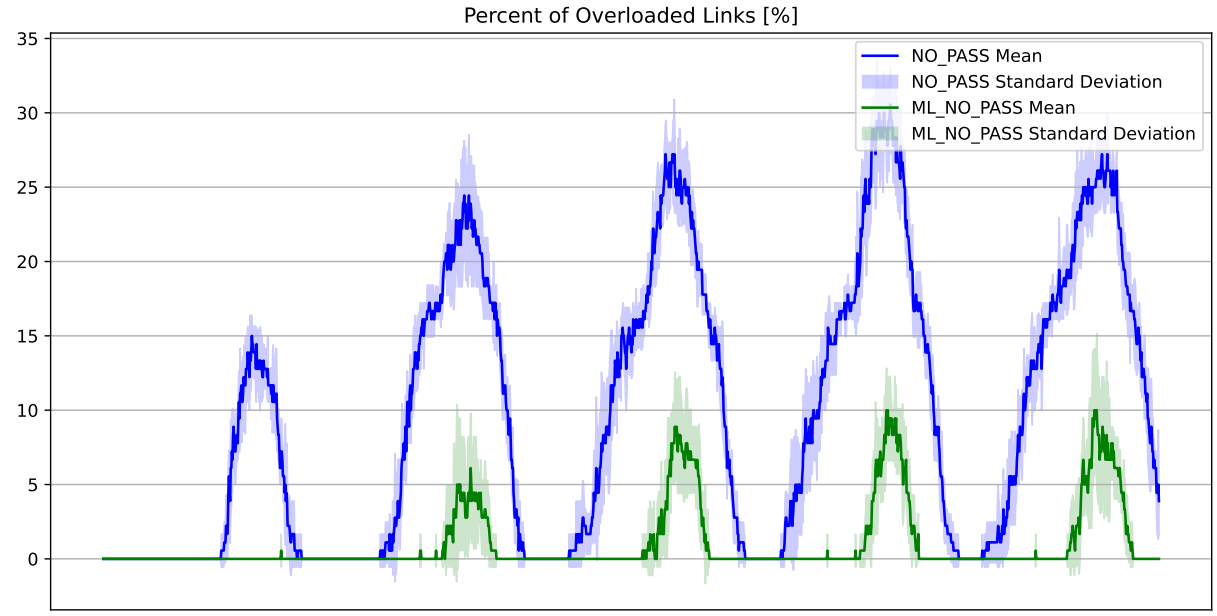

Figure 6.7: Overloaded links for *NO_PASS* and *150* generator rate.

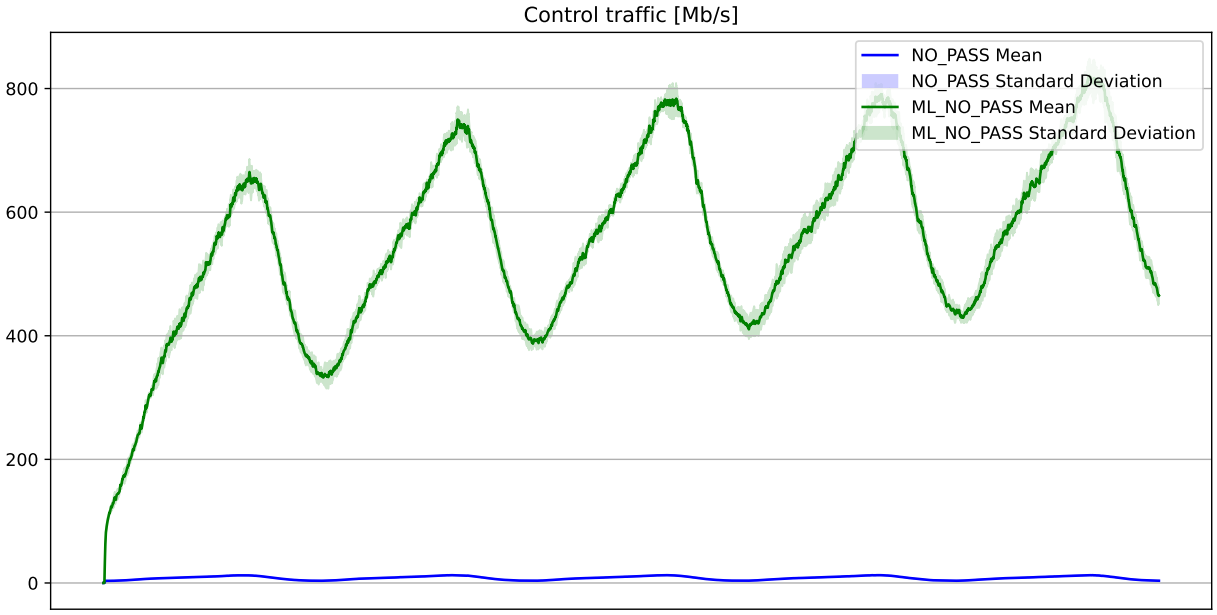

Figure 6.8: Control traffic for *NO_PASS* and *150* generator rate.

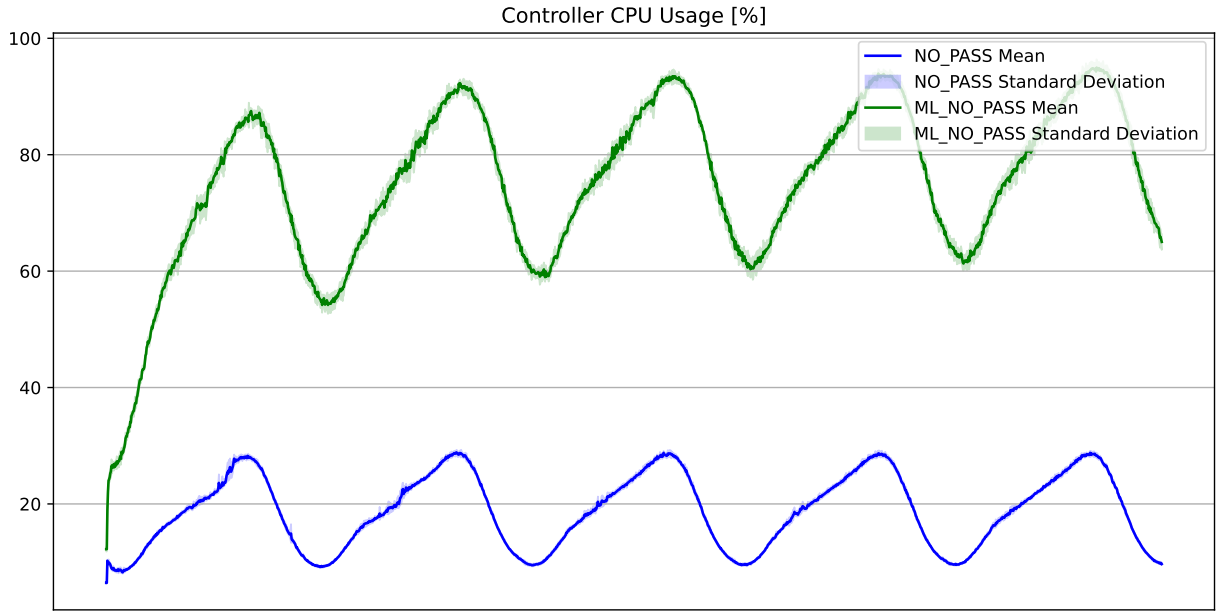

Figure 6.9: Controller CPU usage for *NO_PASS* and *150* generator rate.

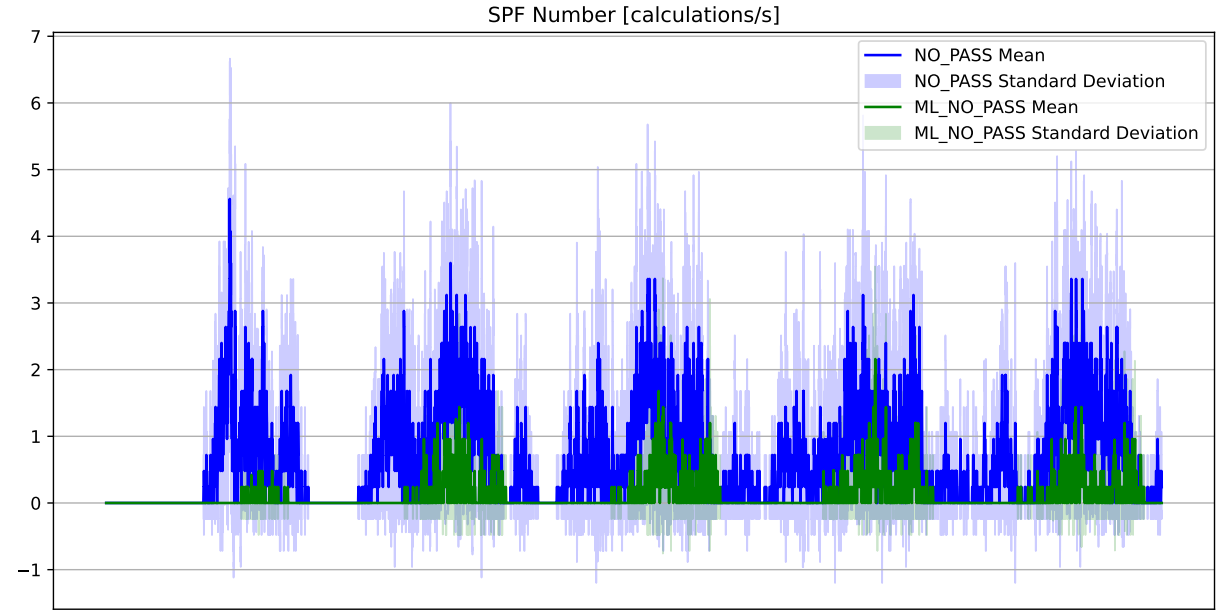

Figure 6.10: SPF calculations for *NO_PASS* and *150* generator rate.

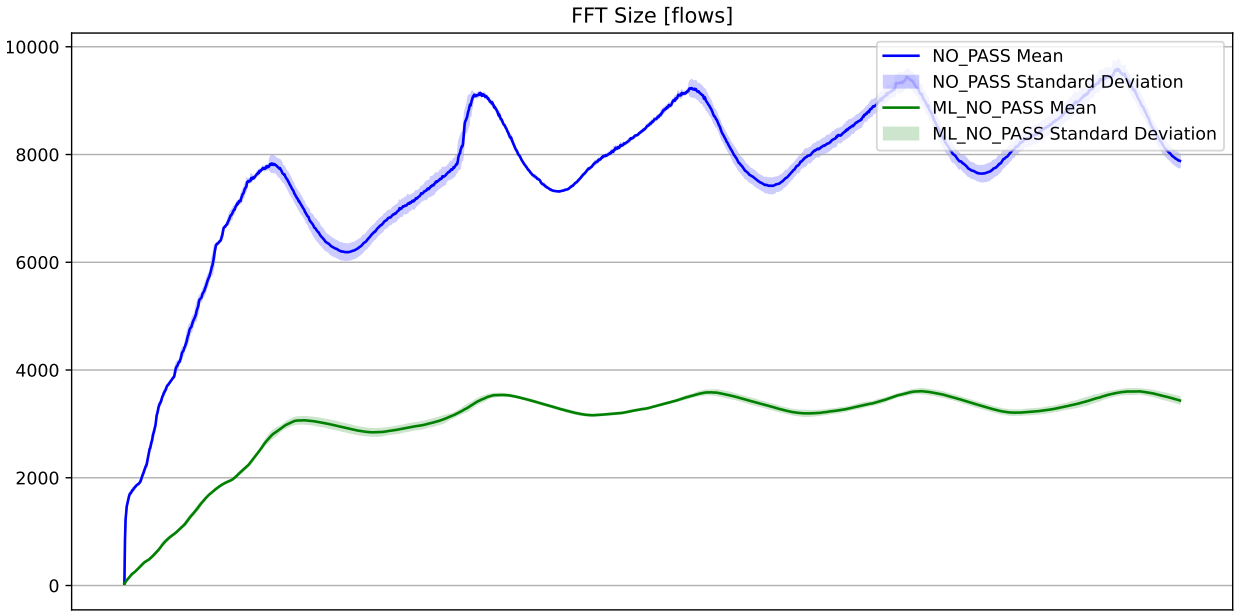

Figure 6.11: FFT size for *NO_PASS* and *150* generator rate.

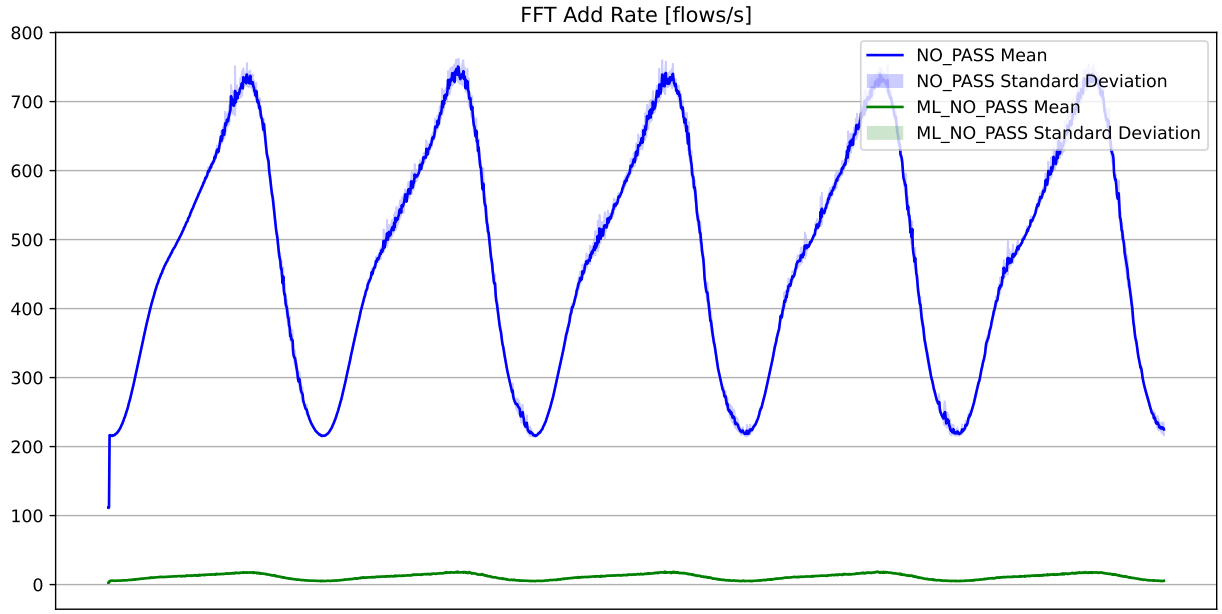

Figure 6.12: FFT add operations for *NO_PASS* and *150* generator rate.

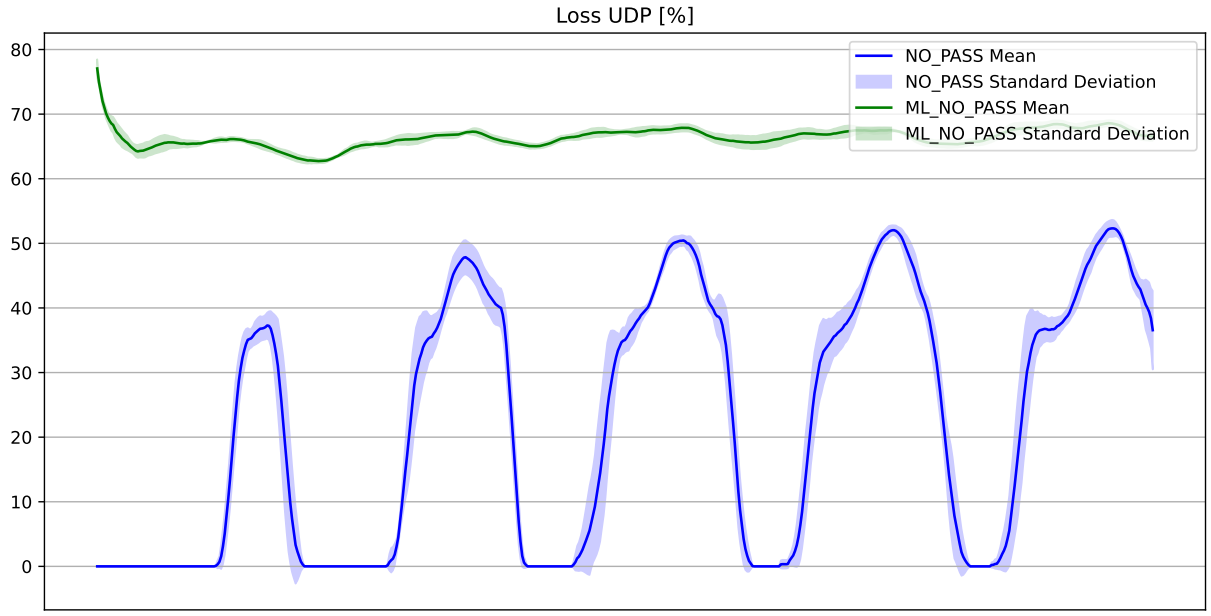

Figure 6.13: UDP losses for *NO_PASS* and *150* generator rate.

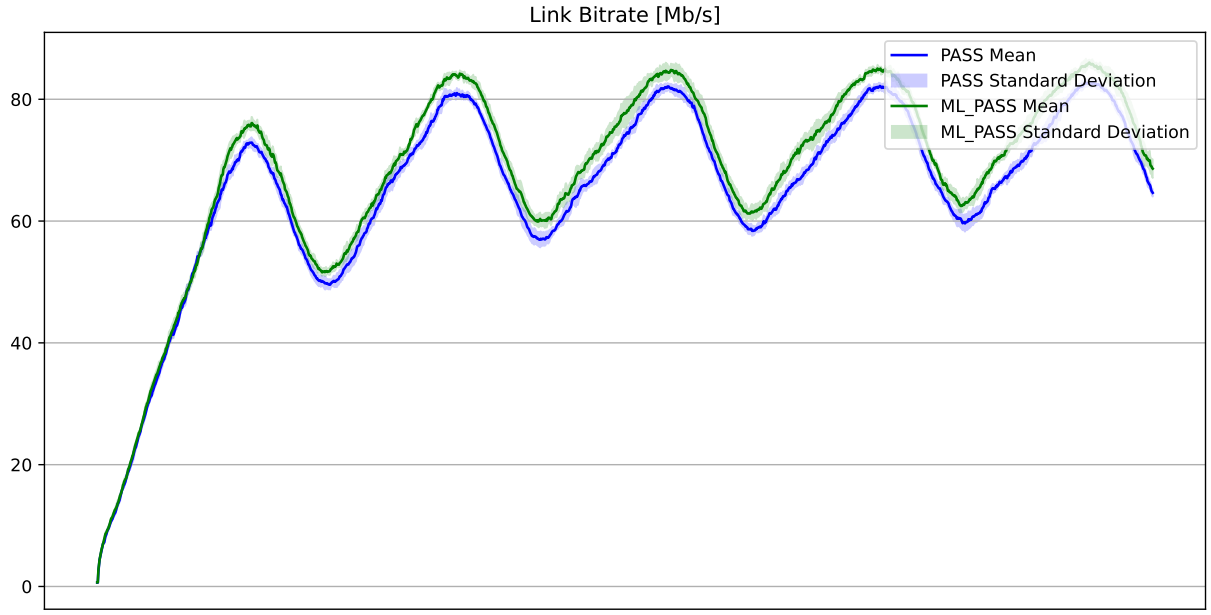

Figure 6.14: Link throughput for *PASS* and *150* generator rate.

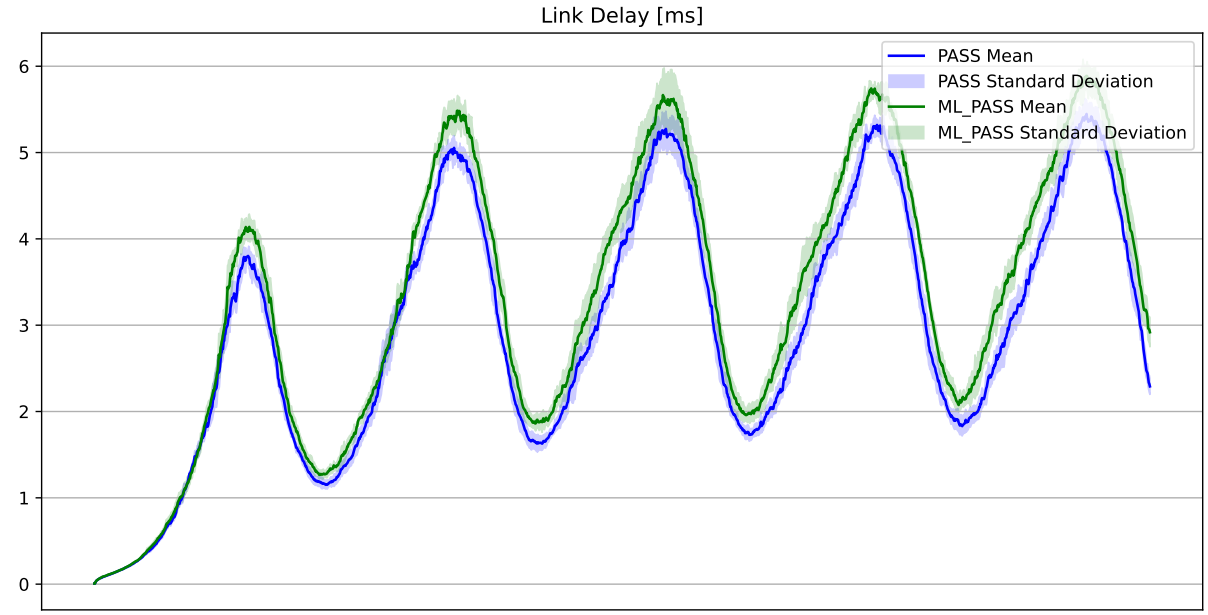

Figure 6.15: Link delay for *PASS* and *150* generator rate.

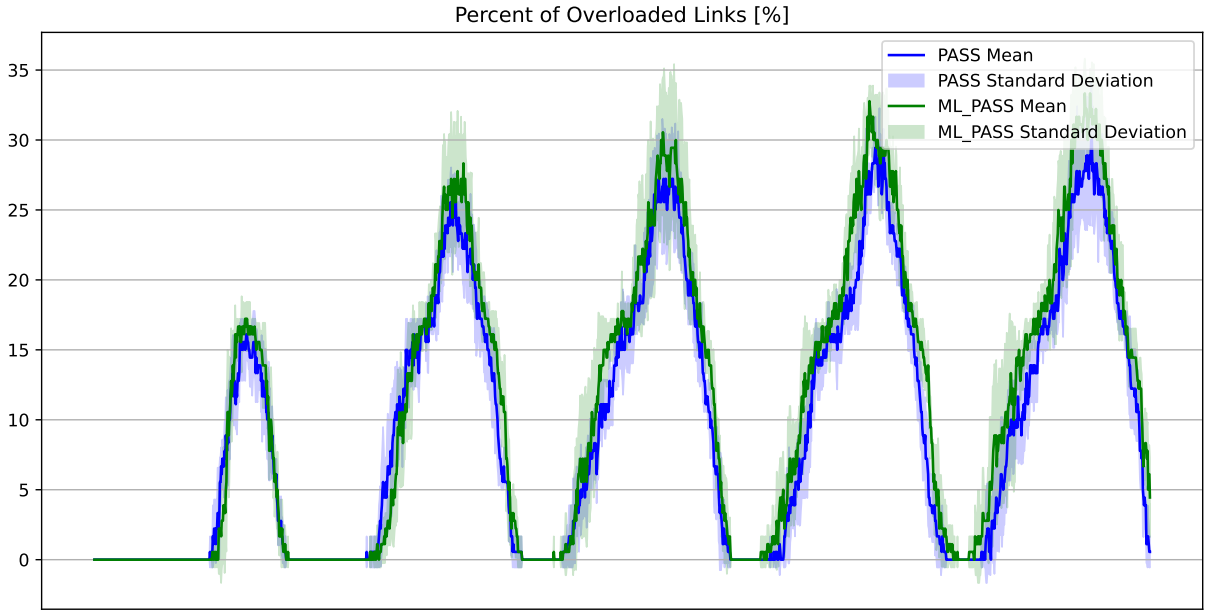

Figure 6.16: Overloaded links for *PASS* and *150* generator rate.

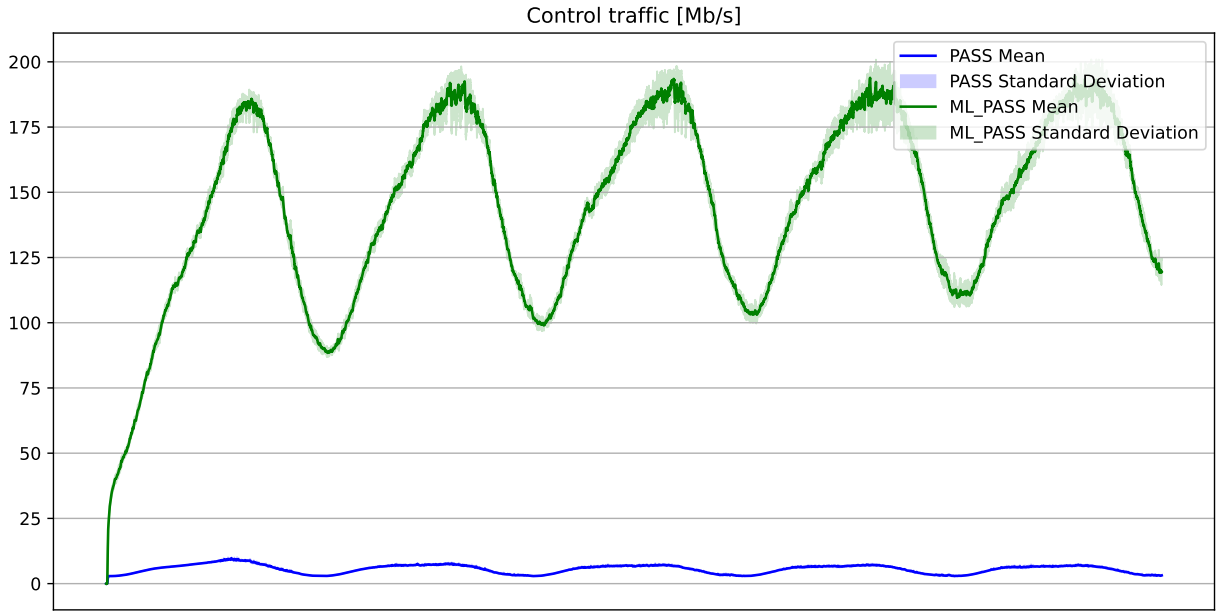

Figure 6.17: Control traffic for *PASS* and *150* generator rate.

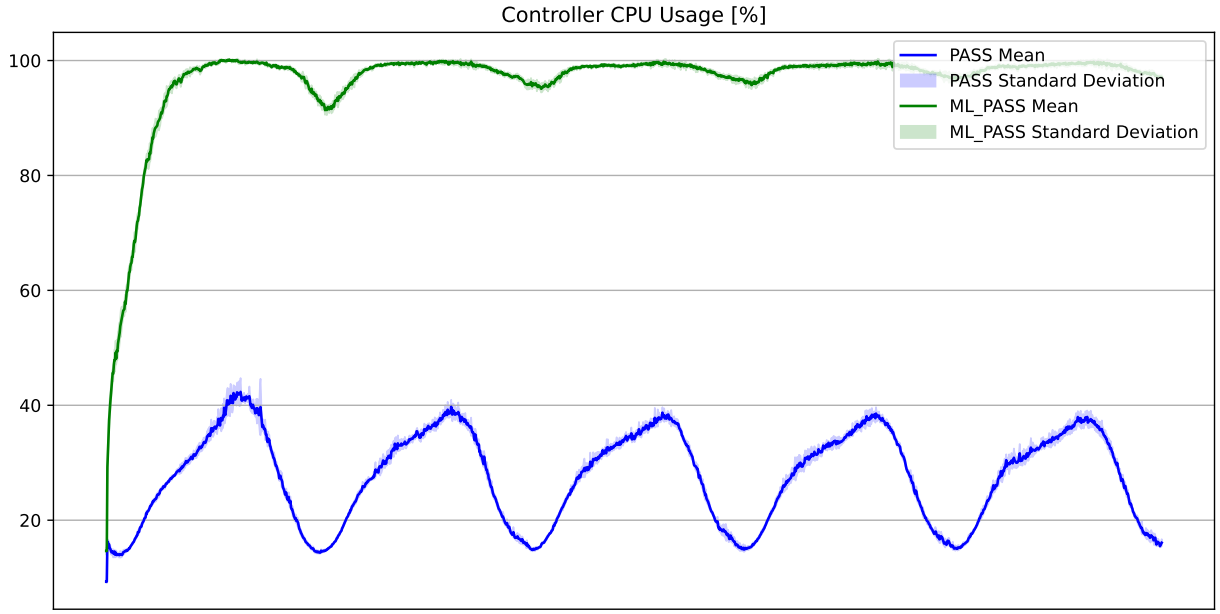

Figure 6.18: Controller CPU usage for *PASS* and *150* generator rate.

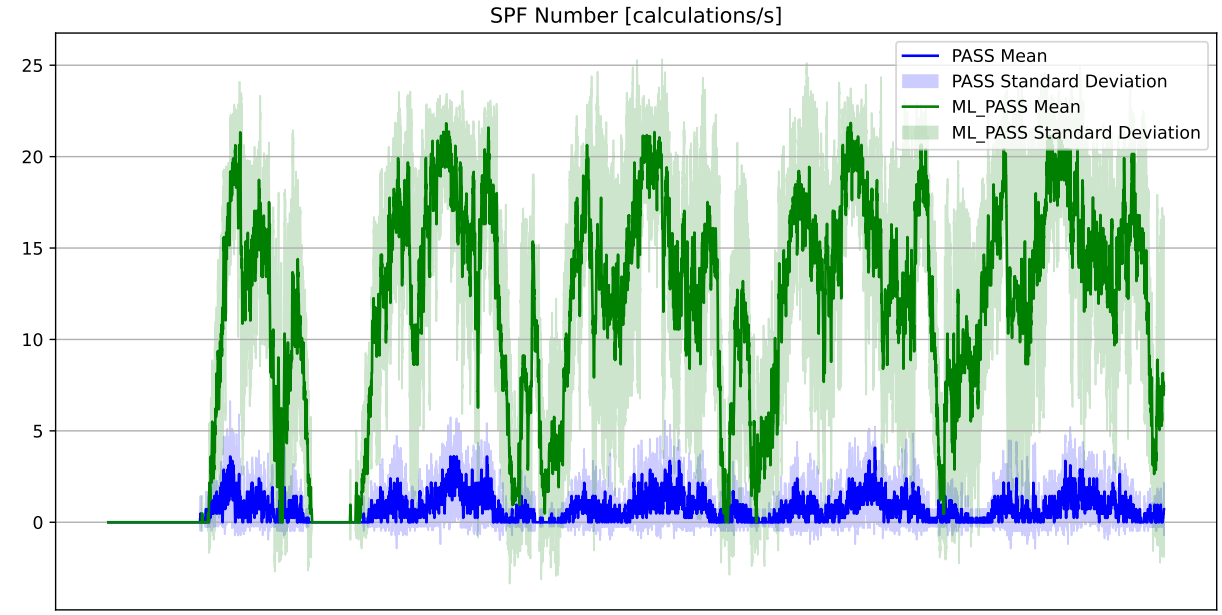

Figure 6.19: SPF calculations for *PASS* and *150* generator rate.

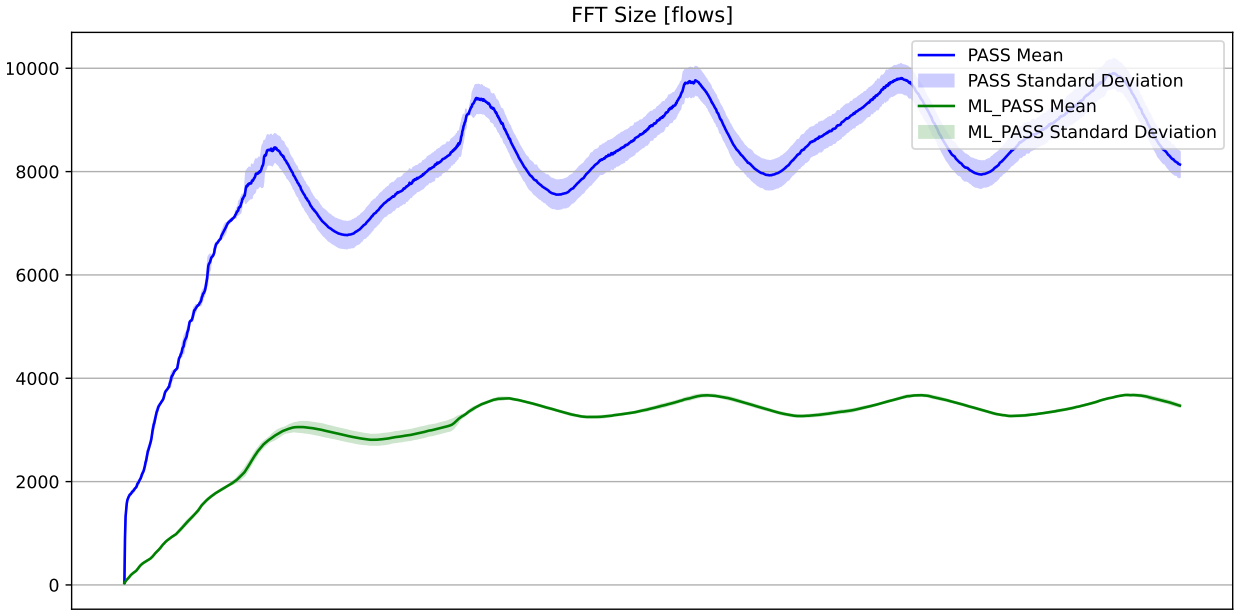

Figure 6.20: FFT size for *PASS* and *150* generator rate.

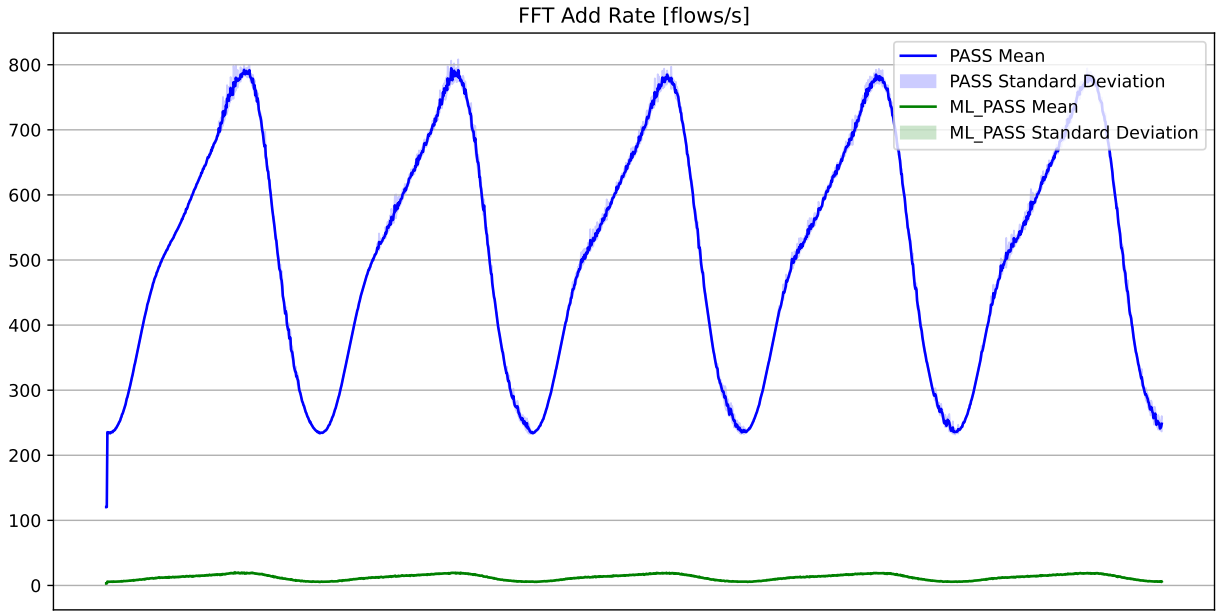

Figure 6.21: FFT add operations for *PASS* and *150* generator rate.

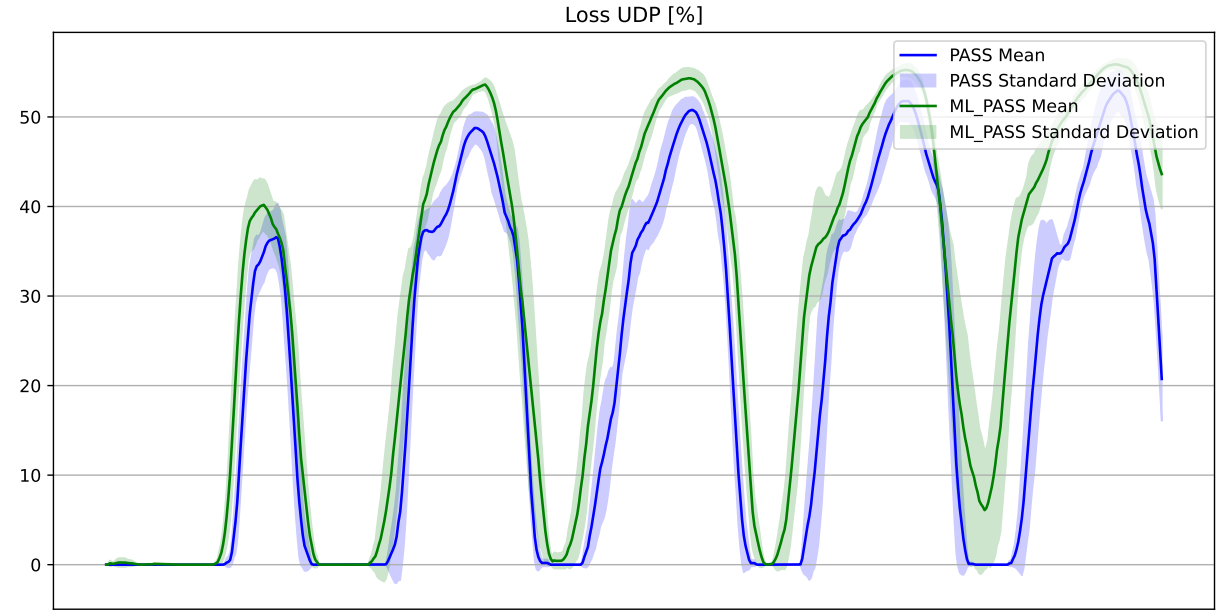

Figure 6.22: UDP losses for *PASS* and *150* generator rate.

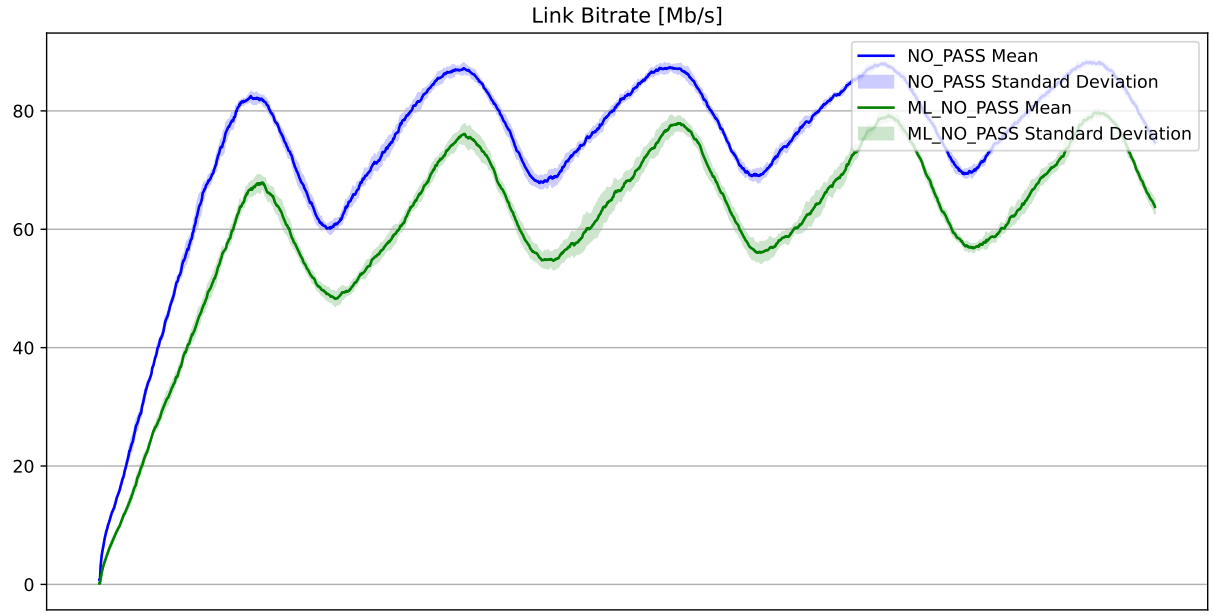

Figure 6.23: Link throughput for *NO_PASS* and *180* generator rate.

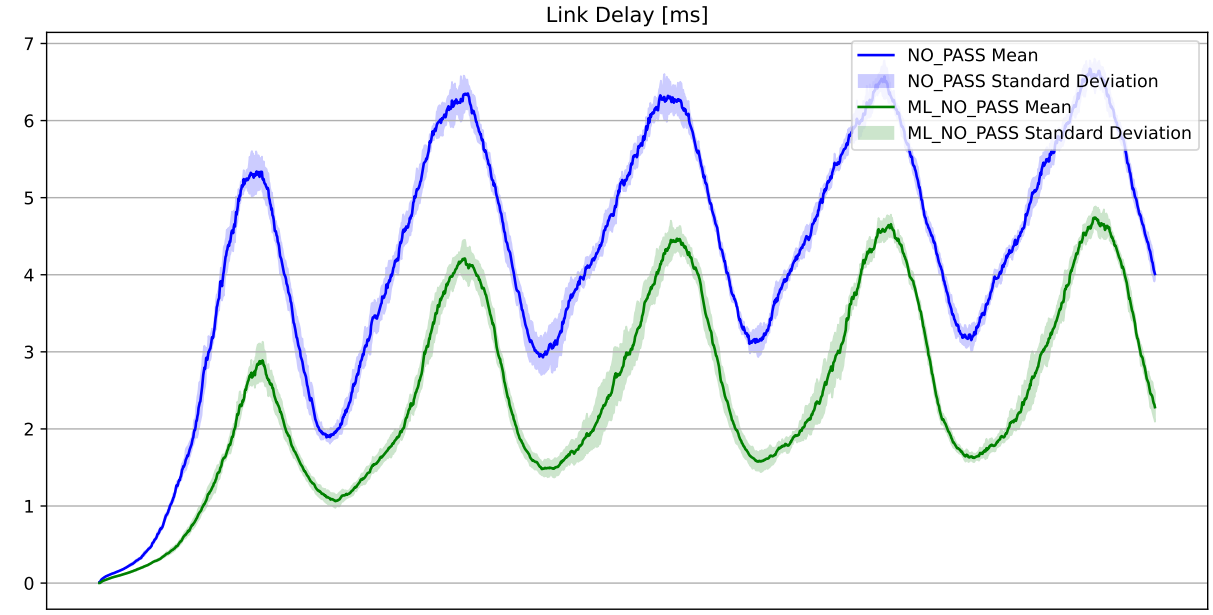

Figure 6.24: Link delay for *NO_PASS* and *180* generator rate.

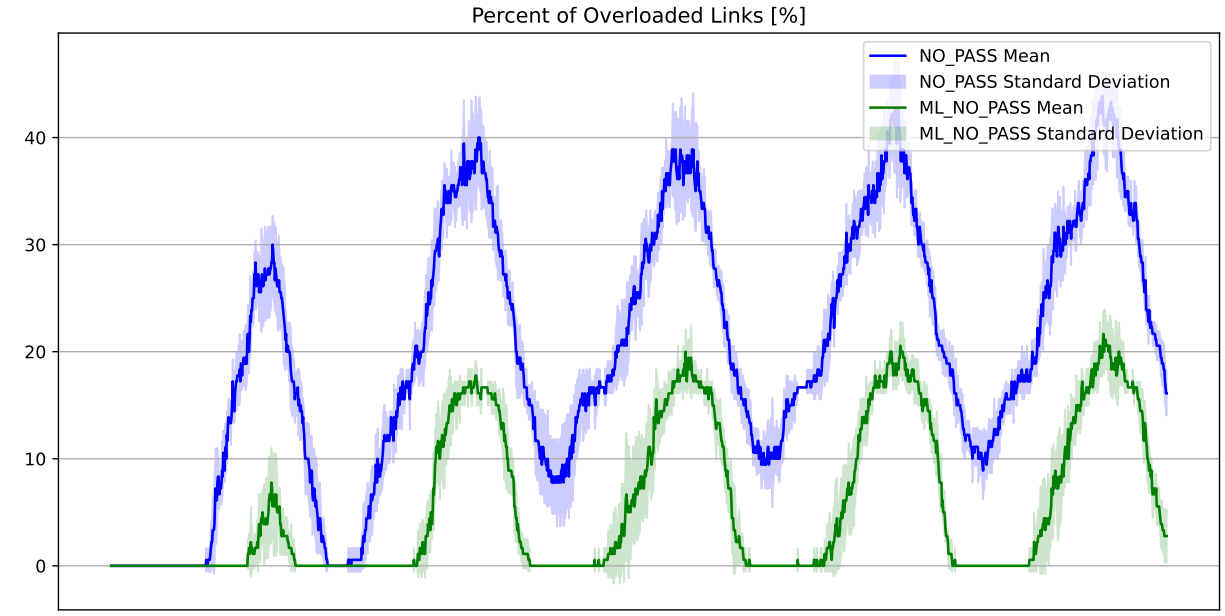

Figure 6.25: Overloaded links for *NO_PASS* and *180* generator rate.

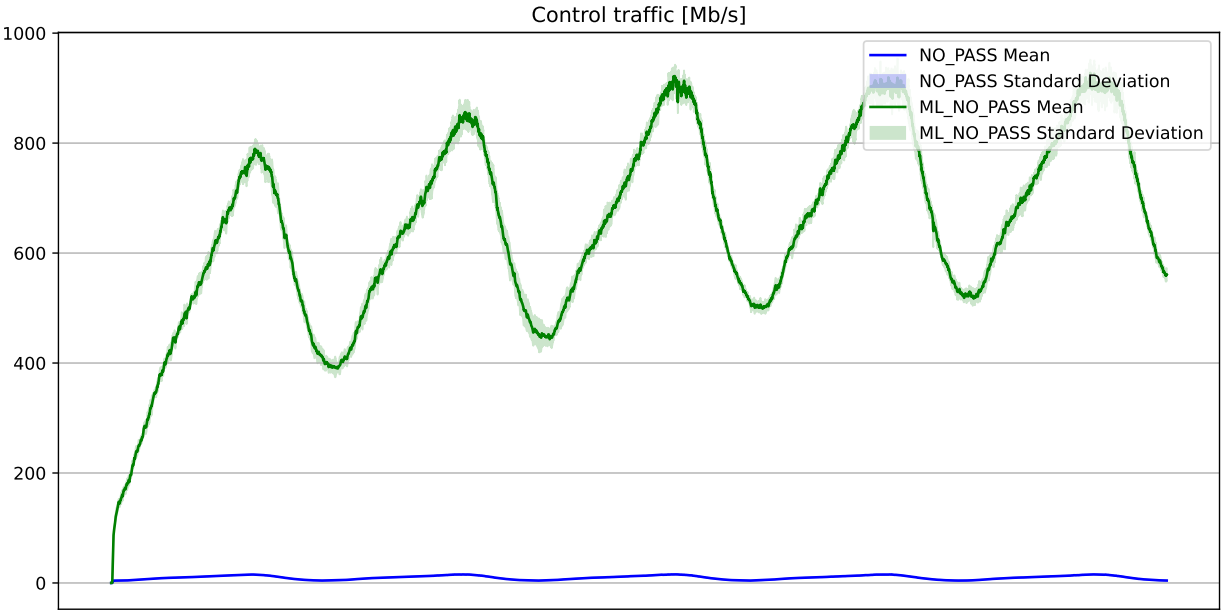

Figure 6.26: Control traffic for *NO_PASS* and *180* generator rate.

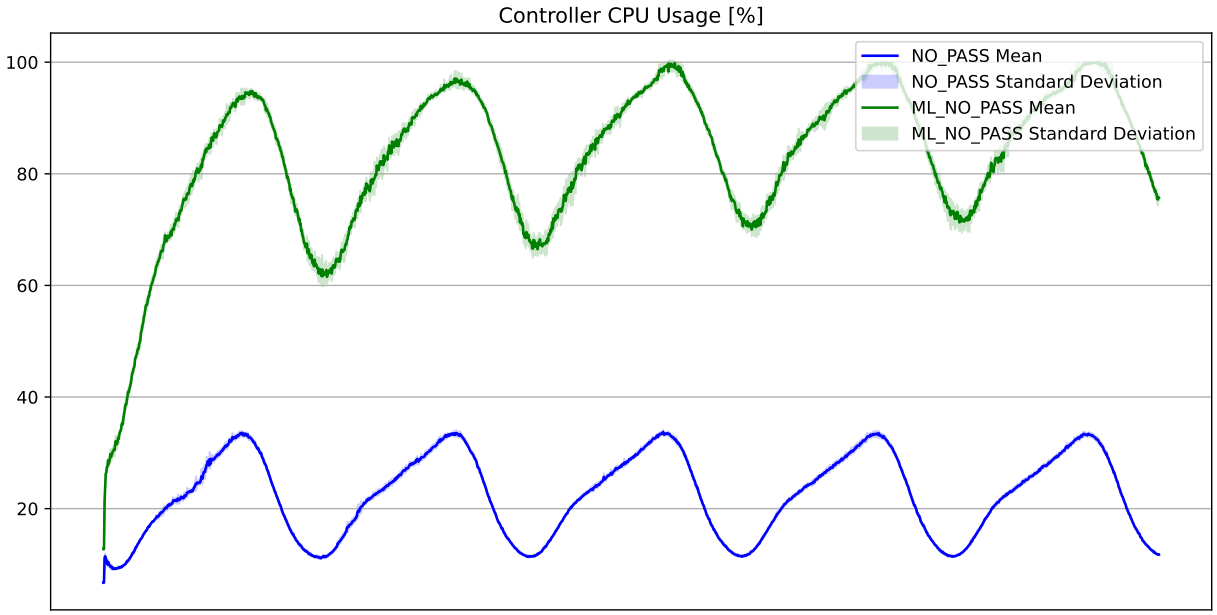

Figure 6.27: Controller CPU usage for *NO_PASS* and *180* generator rate.

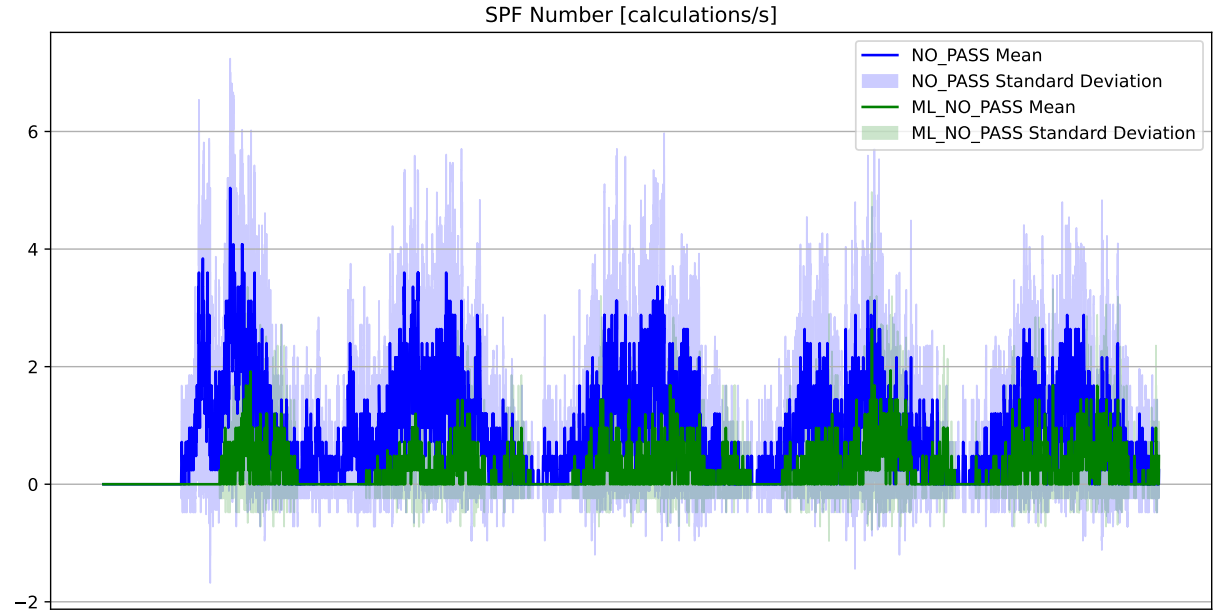

Figure 6.28: SPF calculations for *NO_PASS* and *180* generator rate.

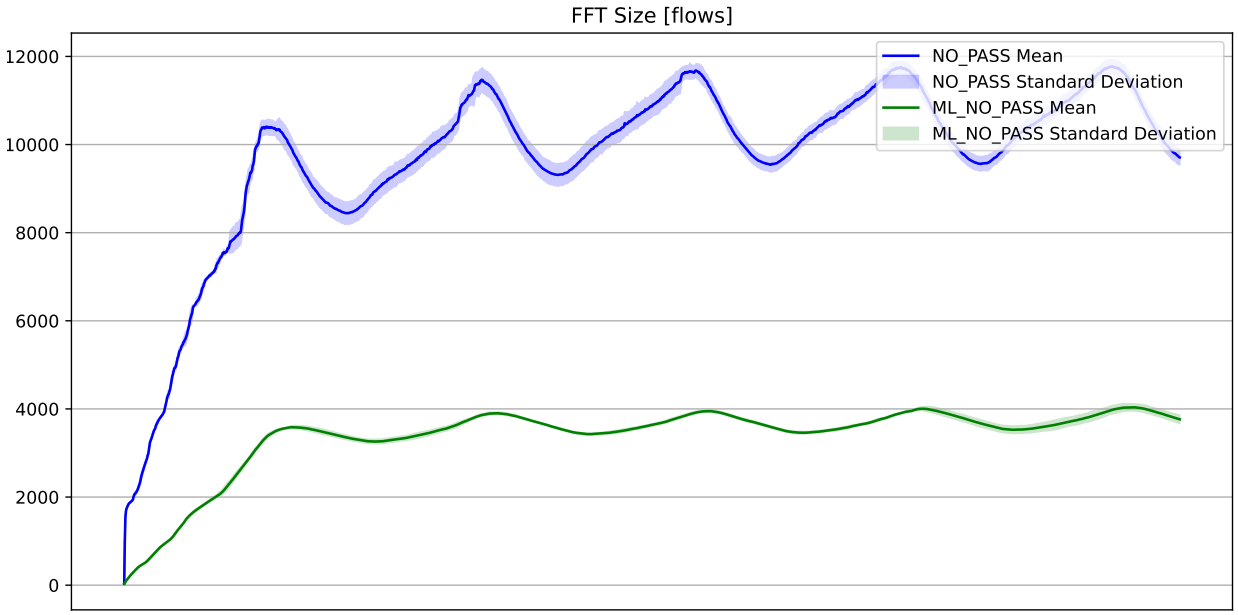

Figure 6.29: FFT size for *NO_PASS* and *180* generator rate.

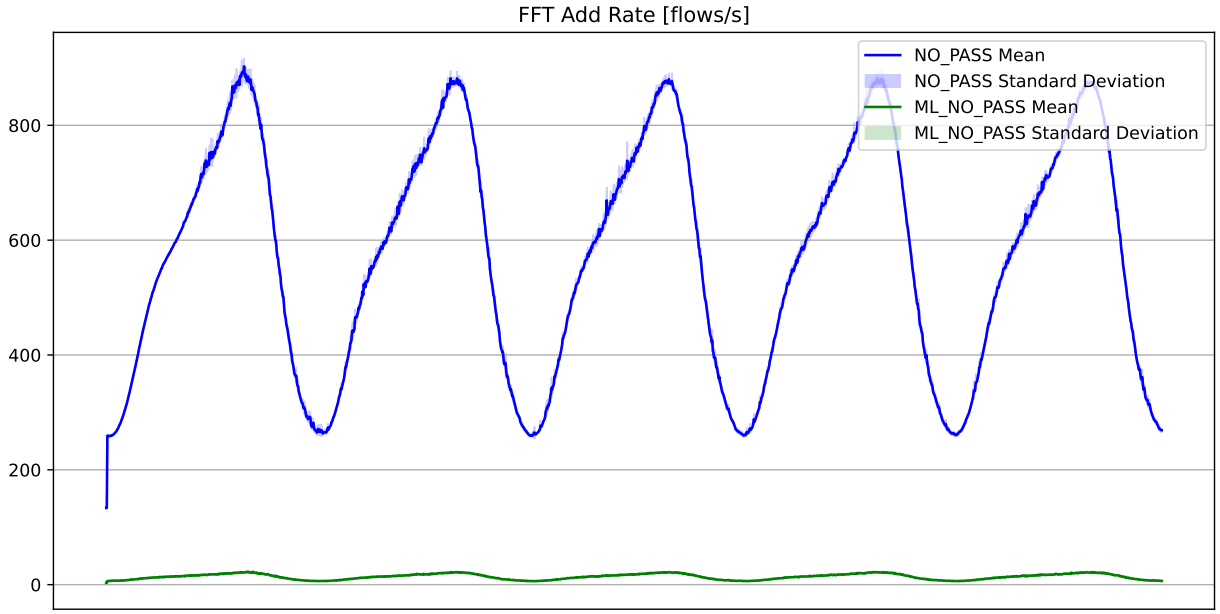

Figure 6.30: FFT add operations for *NO_PASS* and *180* generator rate.

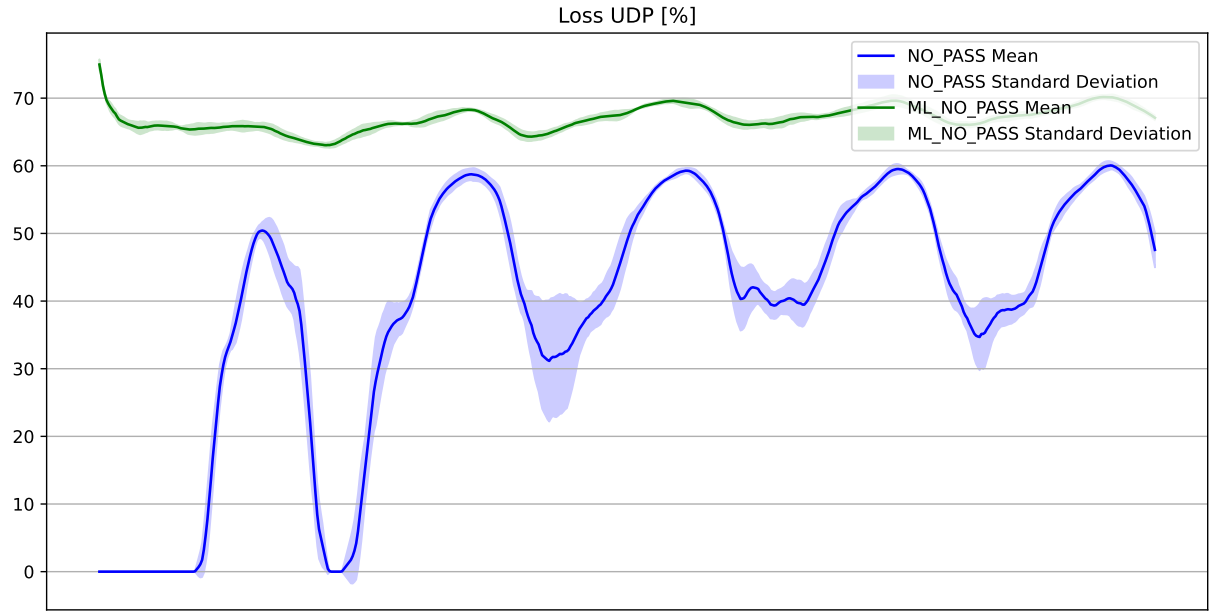

Figure 6.31: UDP losses for *NO_PASS* and *180* generator rate.

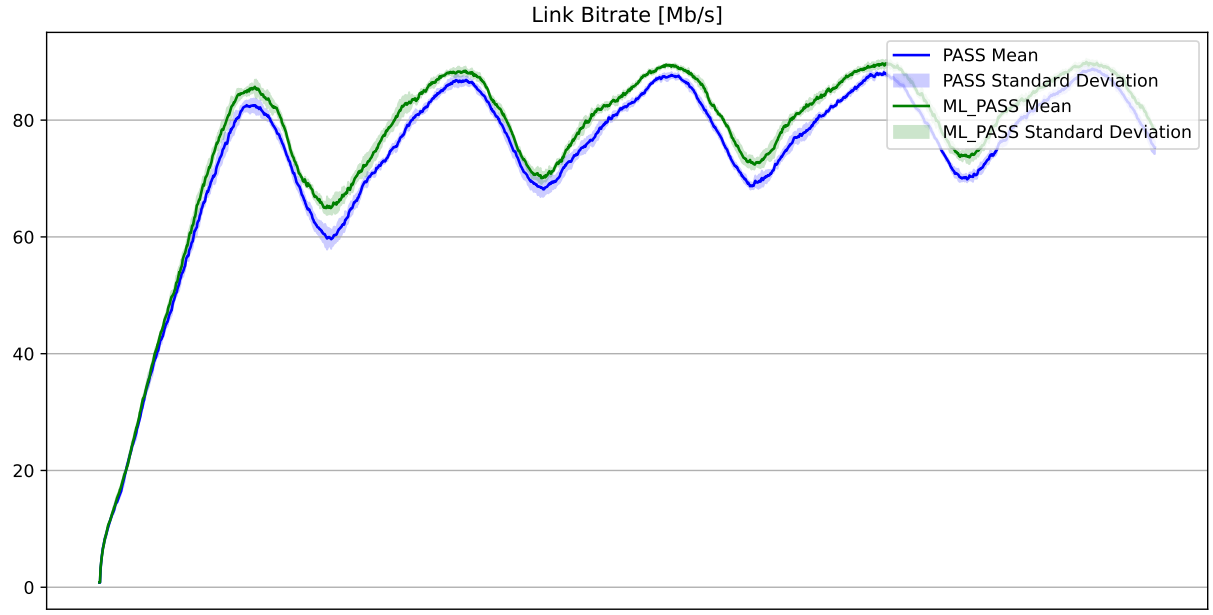

Figure 6.32: Link throughput for *PASS* and *180* generator rate.

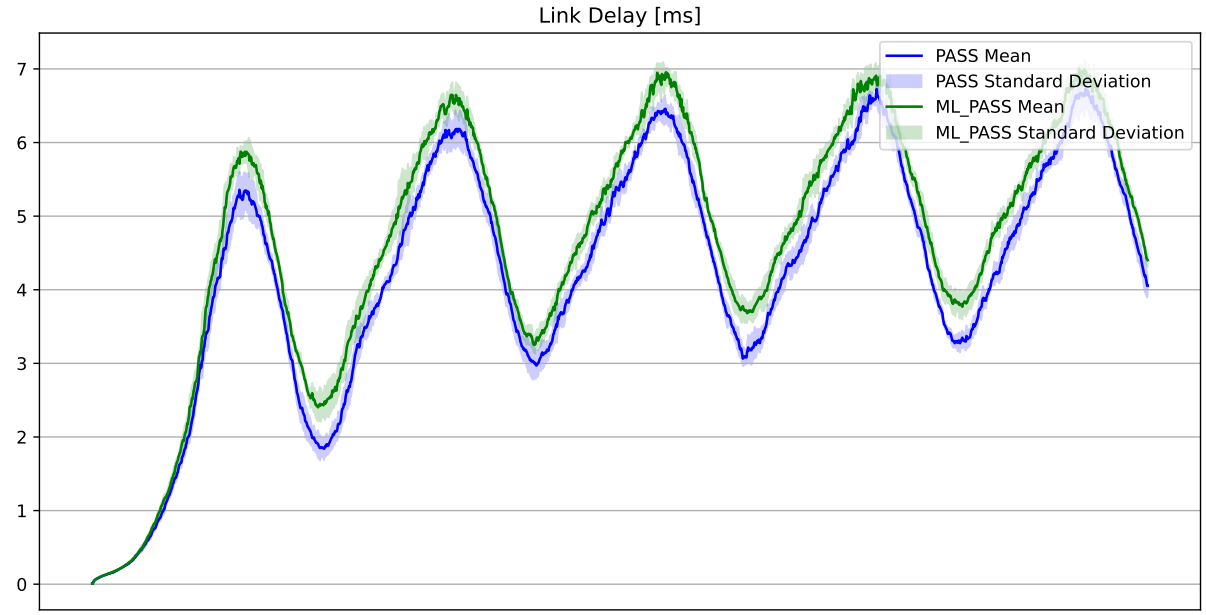

Figure 6.33: Link delay for *PASS* and *180* generator rate.

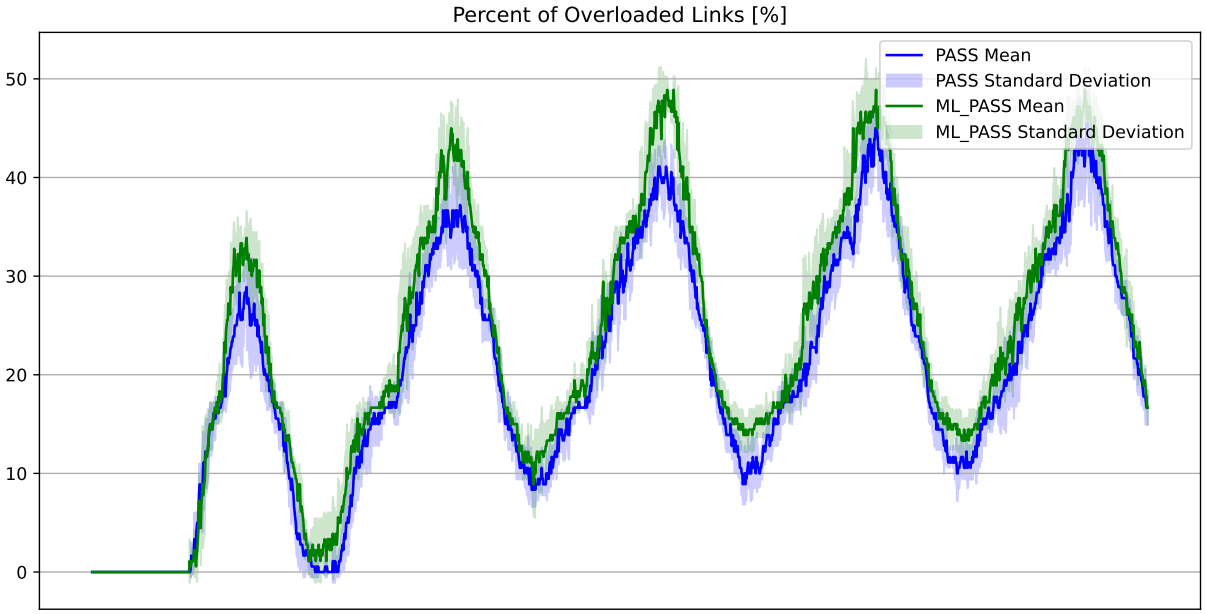

Figure 6.34: Overloaded links for *PASS* and *180* generator rate.

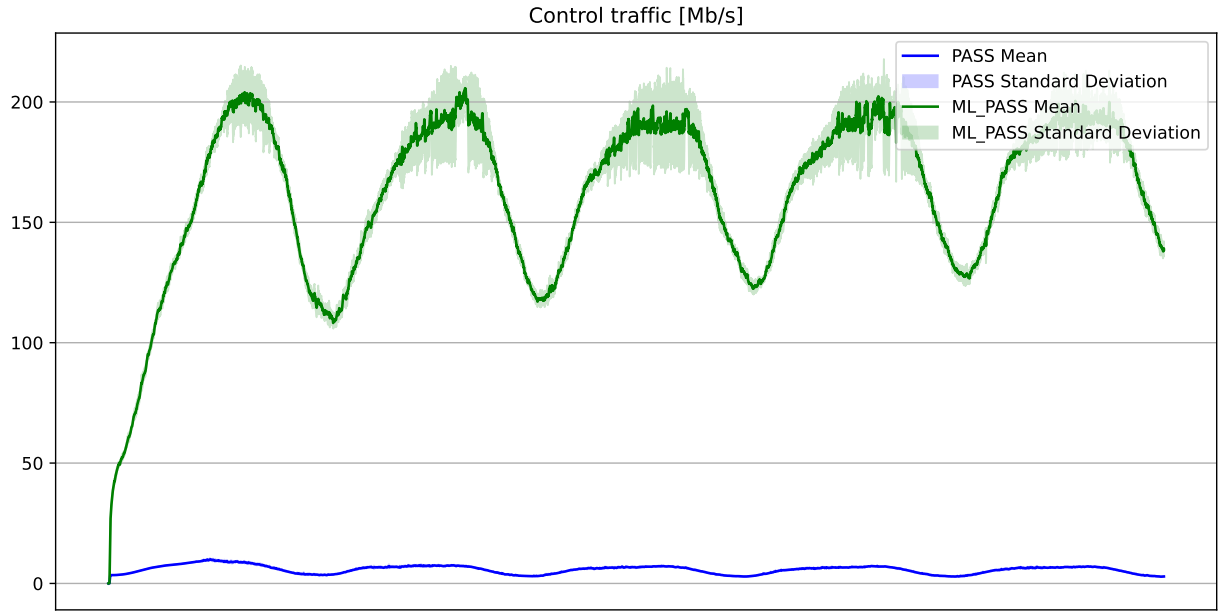

Figure 6.35: Control traffic for *PASS* and *180* generator rate.

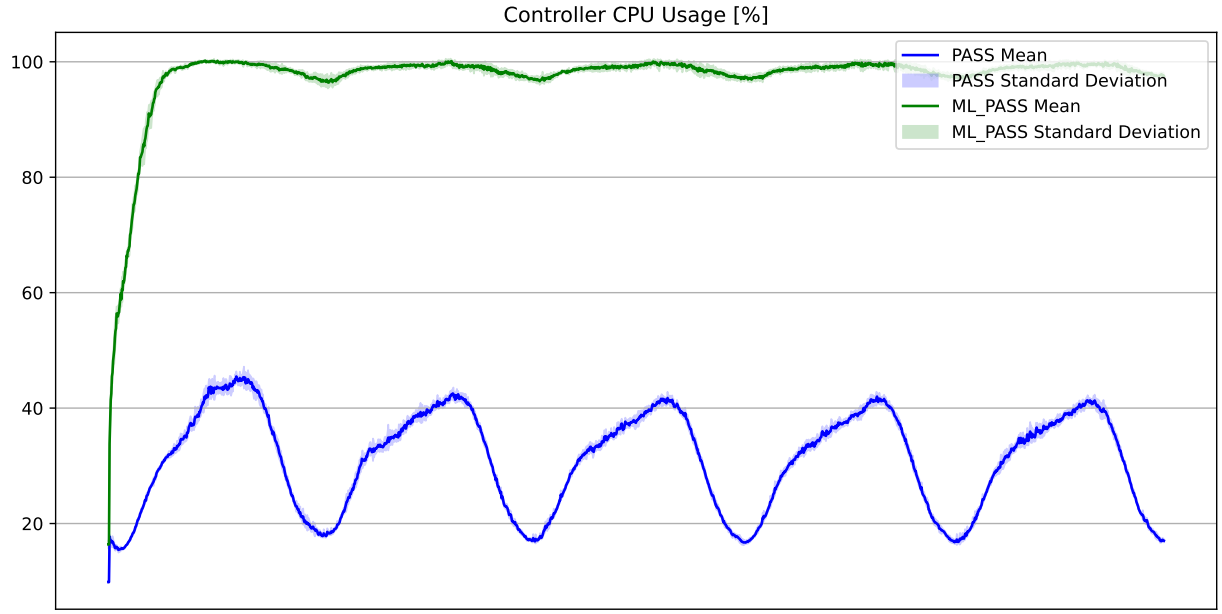

Figure 6.36: Controller CPU usage for *PASS* and *180* generator rate.

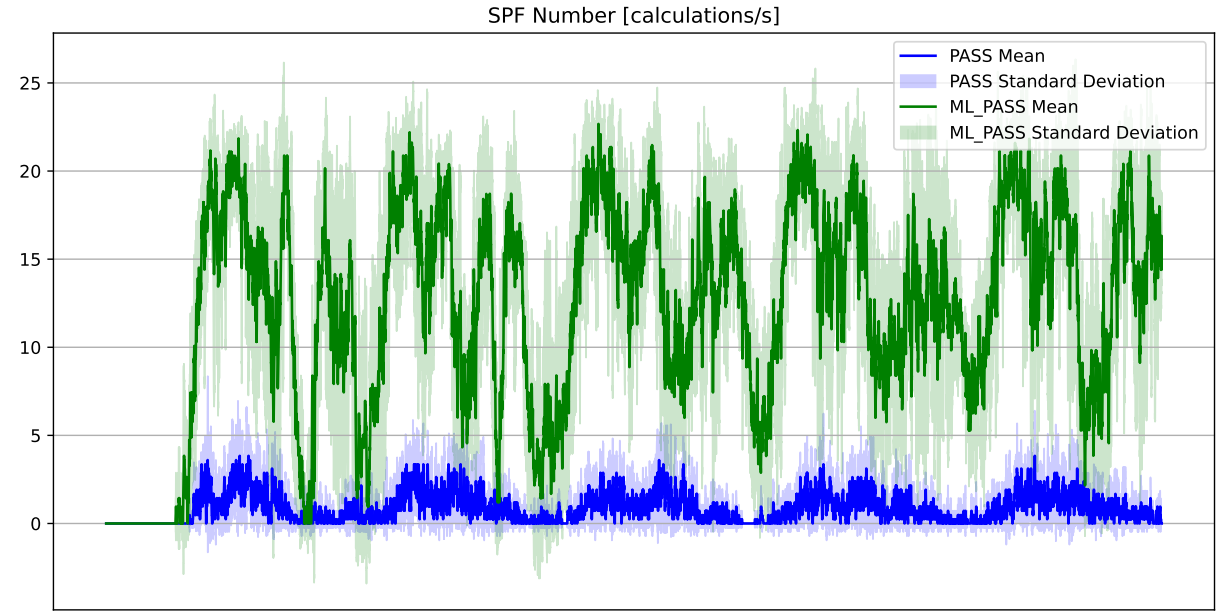

Figure 6.37: SPF calculations for *PASS* and *180* generator rate.

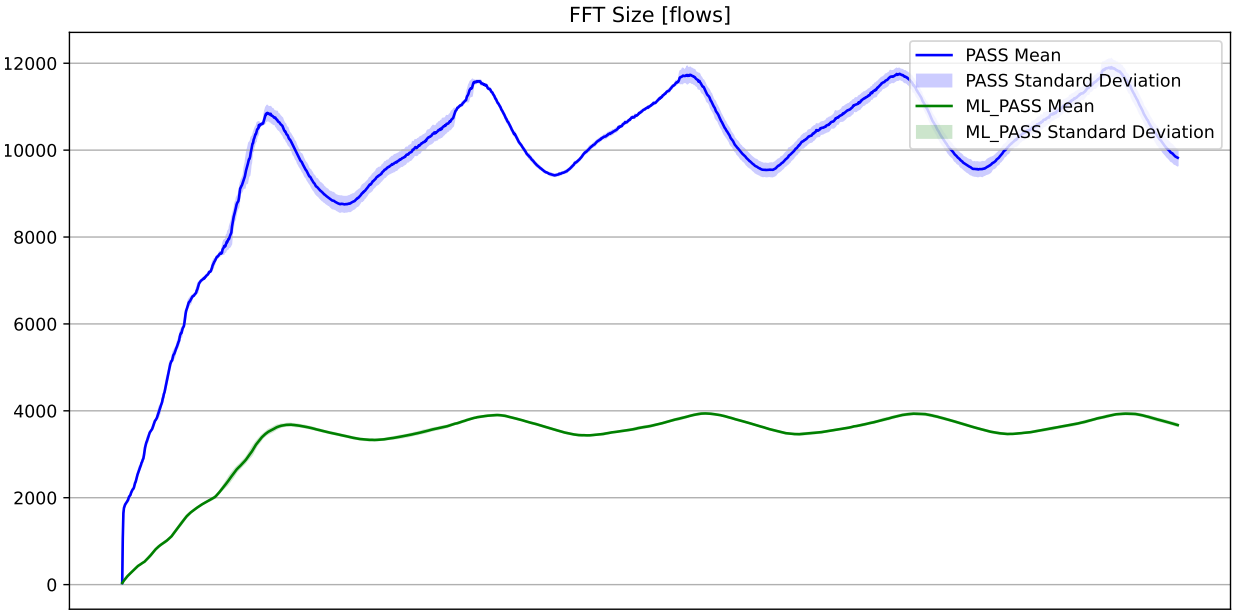

Figure 6.38: FFT size for *PASS* and *180* generator rate.

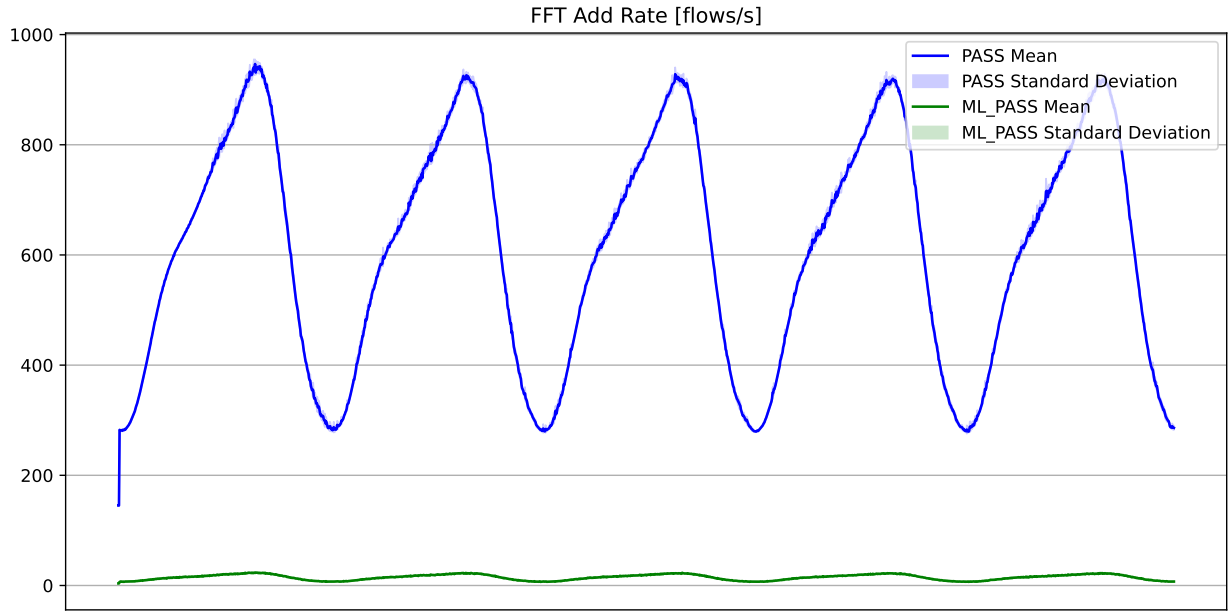

Figure 6.39: FFT add operations for *PASS* and *180* generator rate.

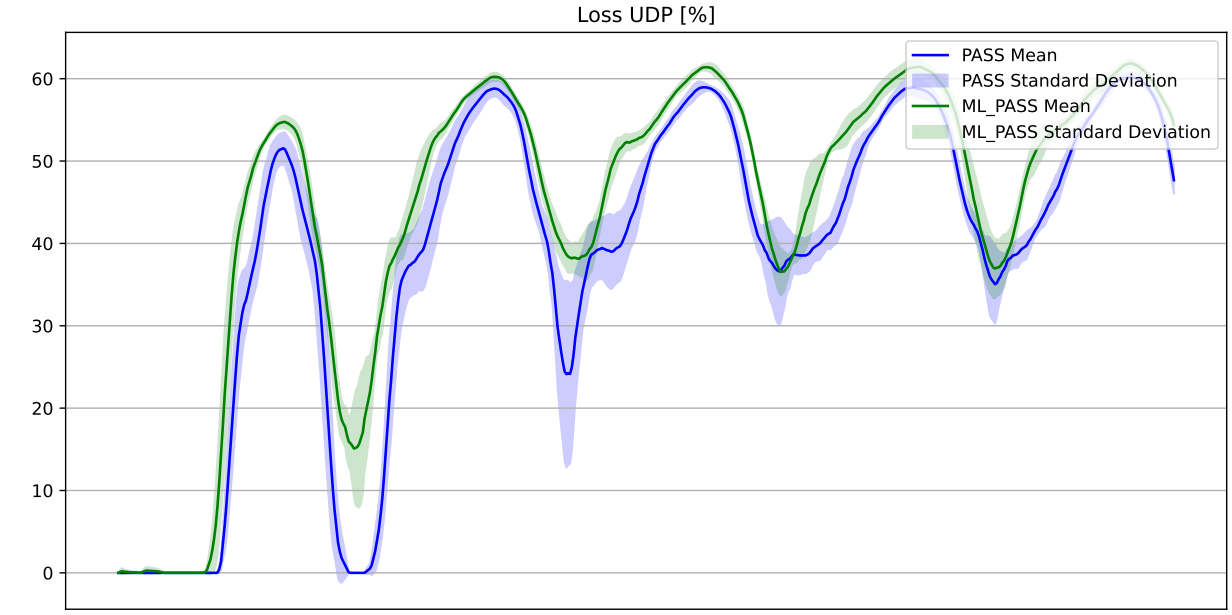

Figure 6.40: UDP losses for *PASS* and *180* generator rate.

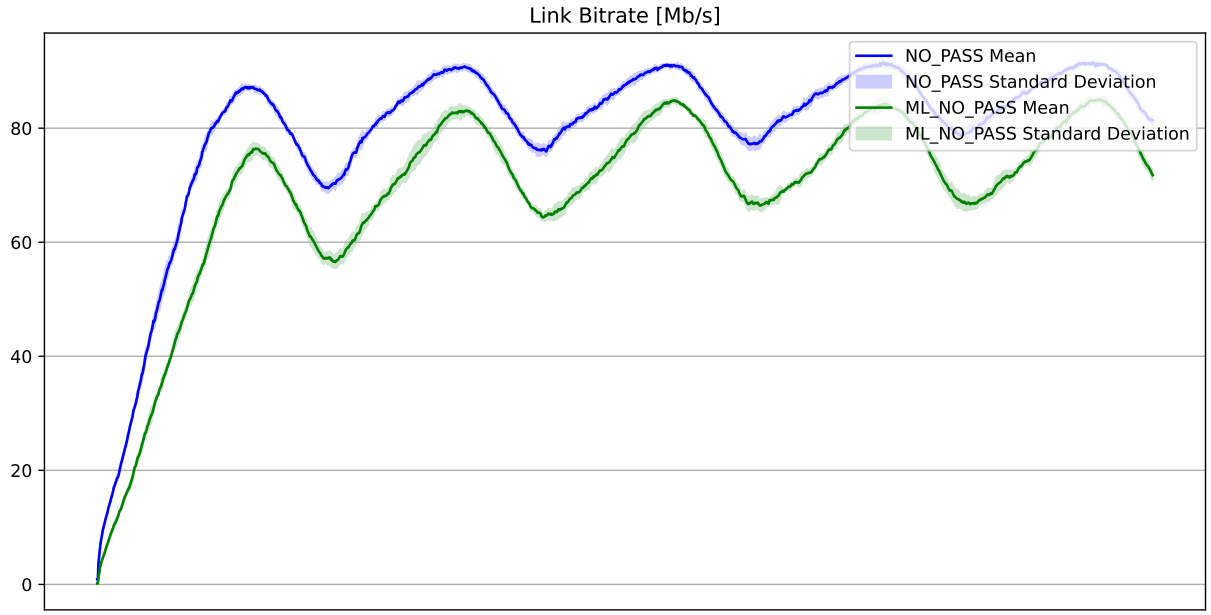

Figure 6.41: Link throughput for *NO_PASS* and *210* generator rate.

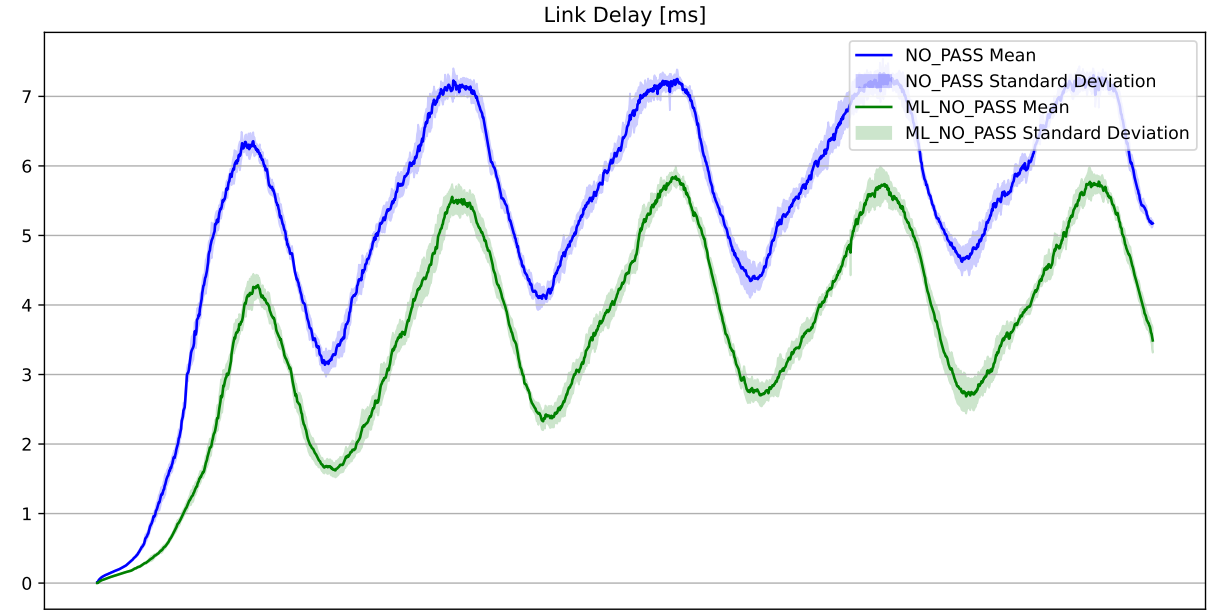

Figure 6.42: Link delay for *NO_PASS* and *210* generator rate.

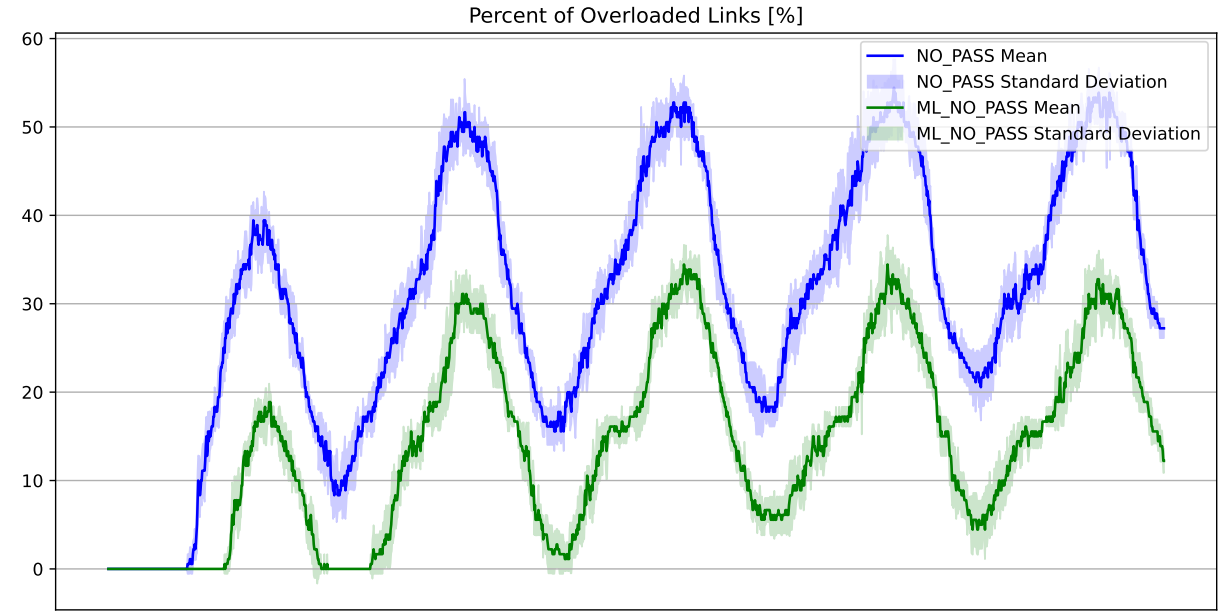

Figure 6.43: Overloaded links for *NO_PASS* and *210* generator rate.

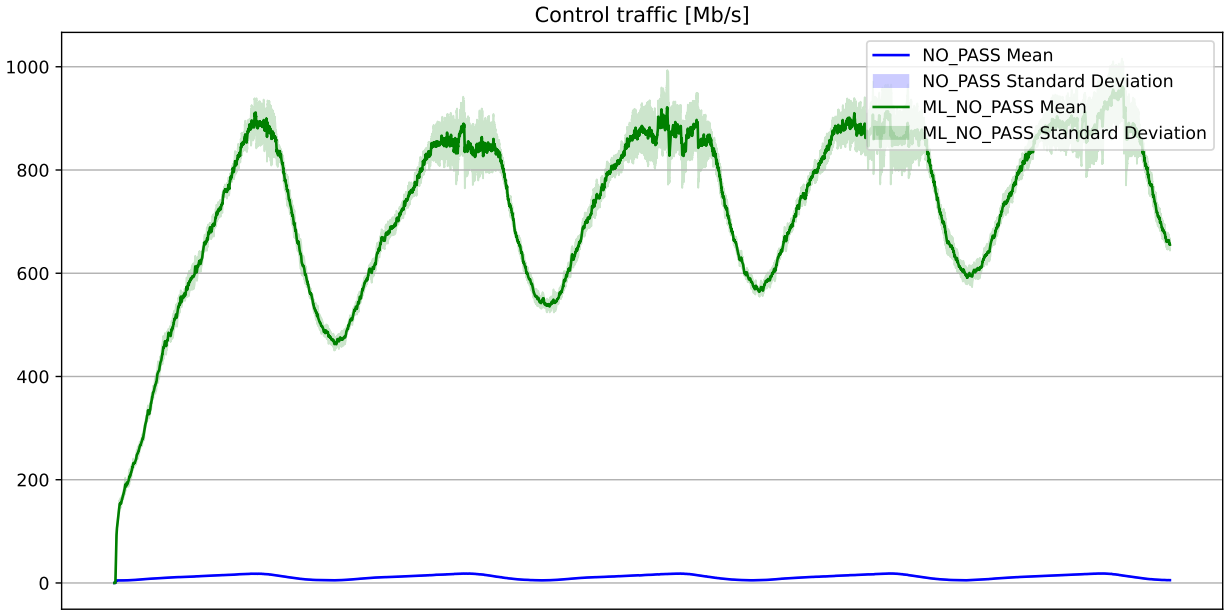

Figure 6.44: Control traffic for *NO_PASS* and *210* generator rate.

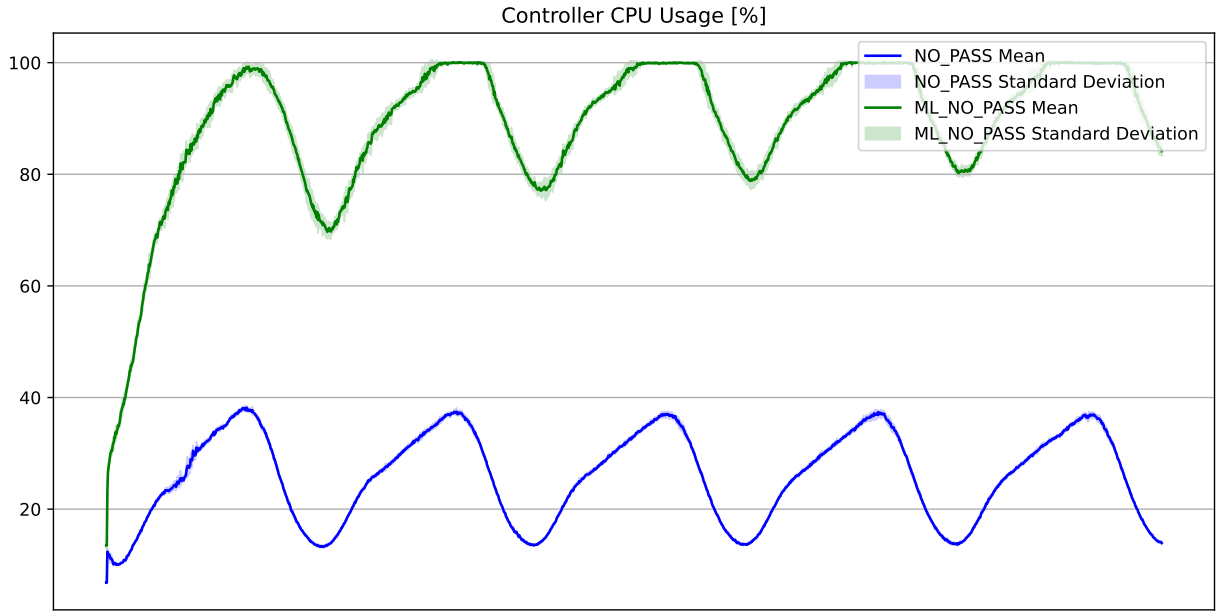

Figure 6.45: Controller CPU usage for *NO_PASS* and *210* generator rate.

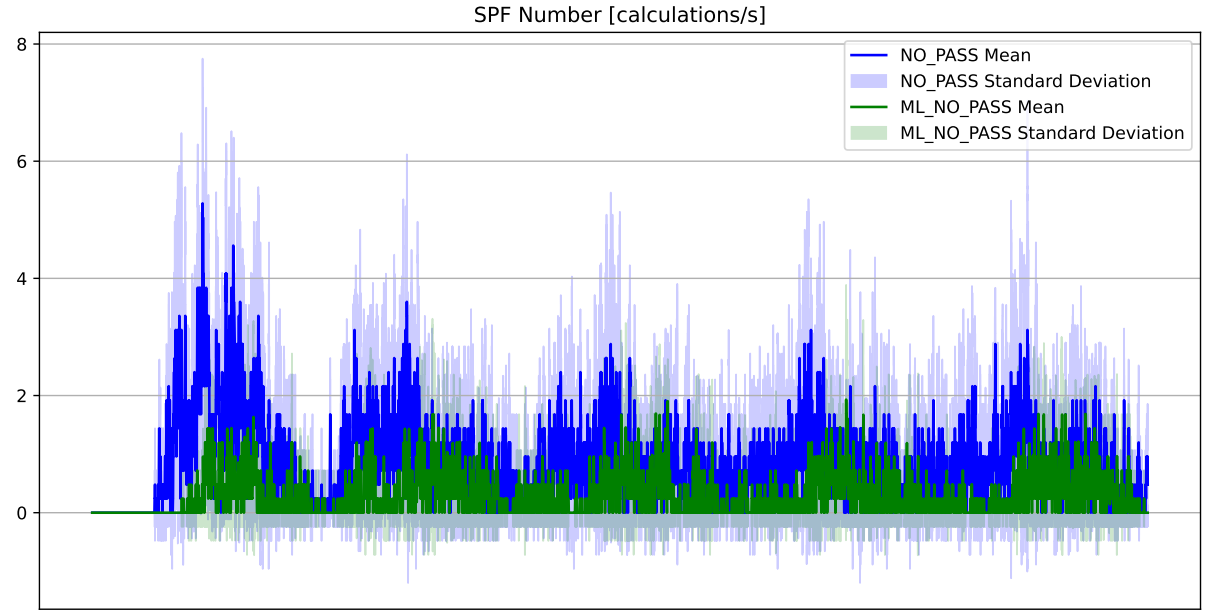

Figure 6.46: SPF calculations for *NO_PASS* and *210* generator rate.

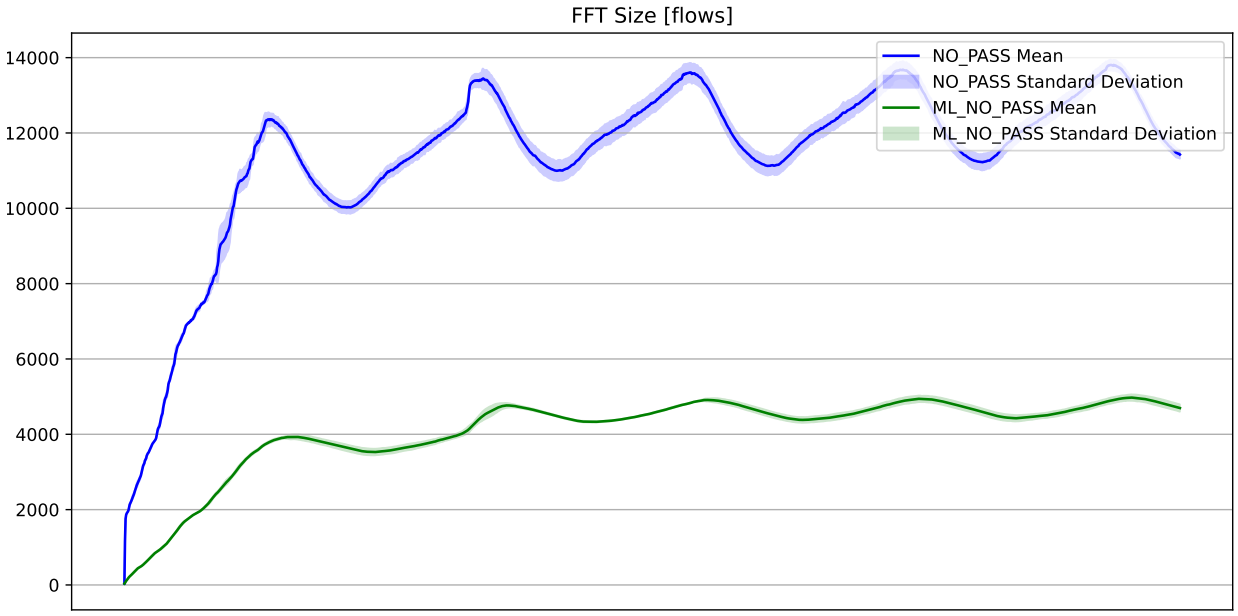

Figure 6.47: FFT size for *NO_PASS* and *210* generator rate.

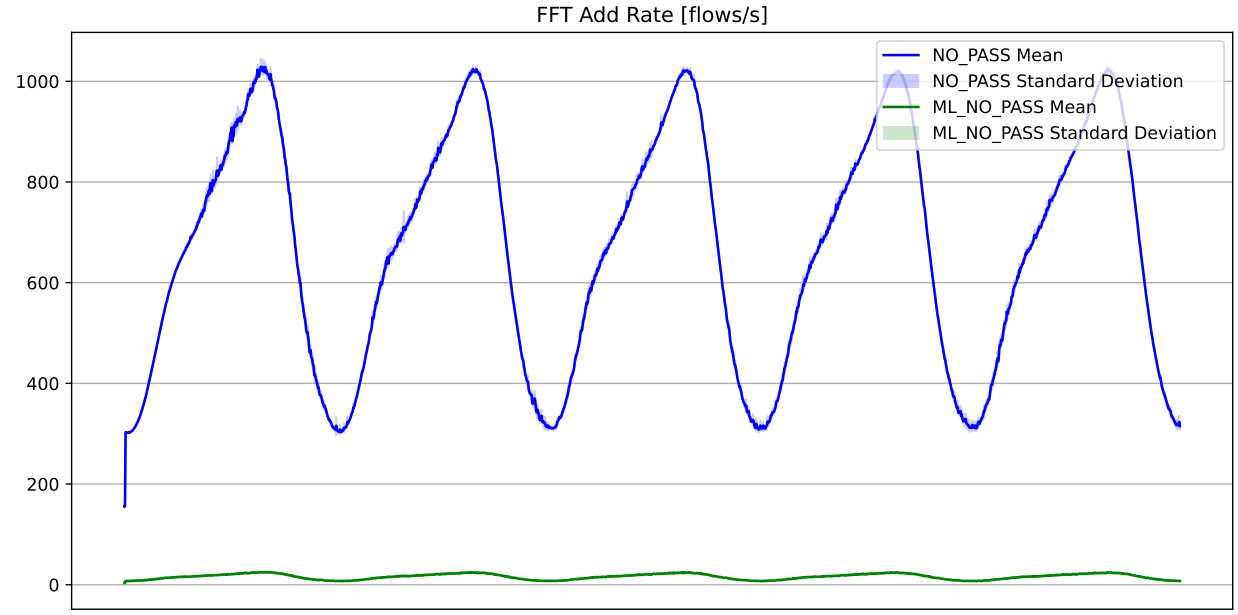

Figure 6.48: FFT add operations for *NO_PASS* and *210* generator rate.

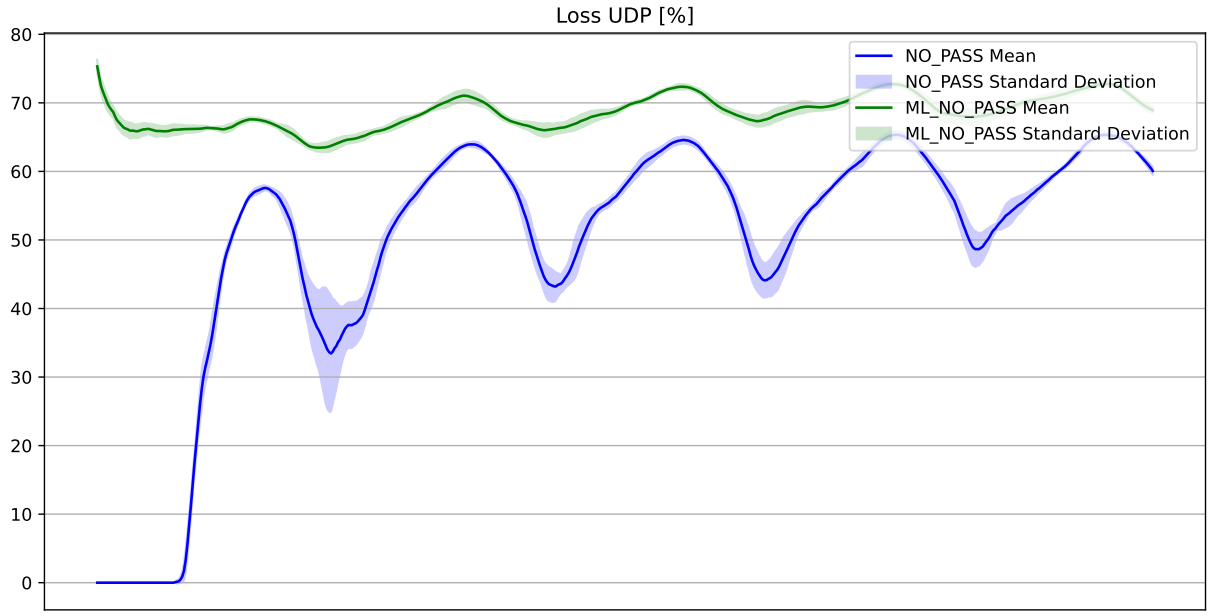

Figure 6.49: UDP losses for *NO_PASS* and *210* generator rate.

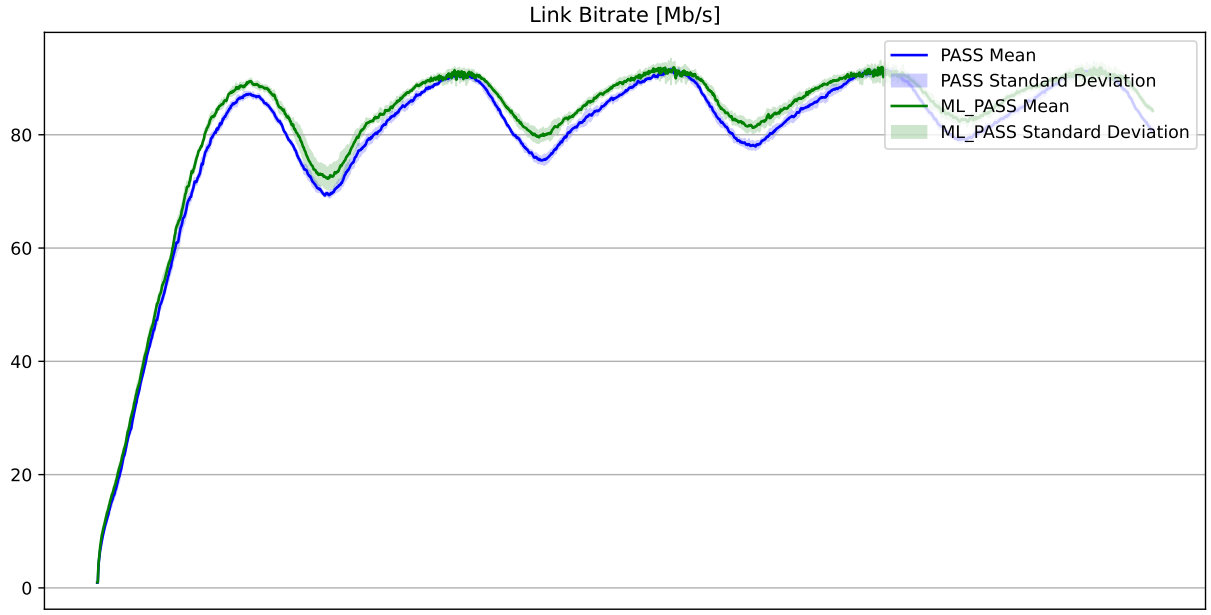

Figure 6.50: Link throughput for *PASS* and *210* generator rate.

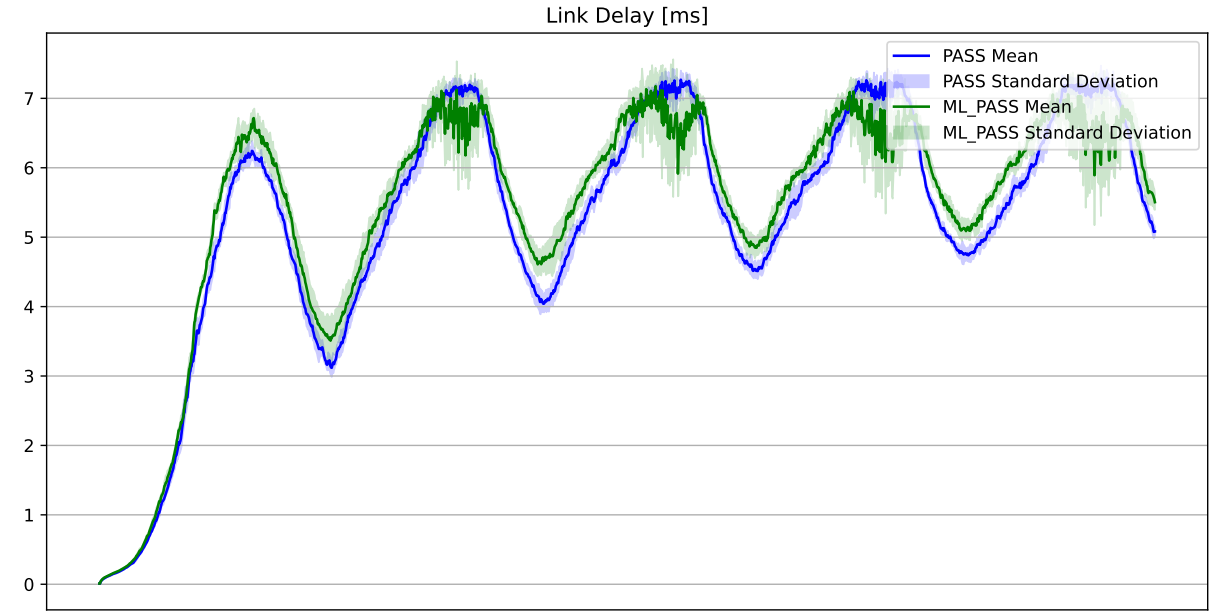

Figure 6.51: Link delay for *PASS* and *210* generator rate.

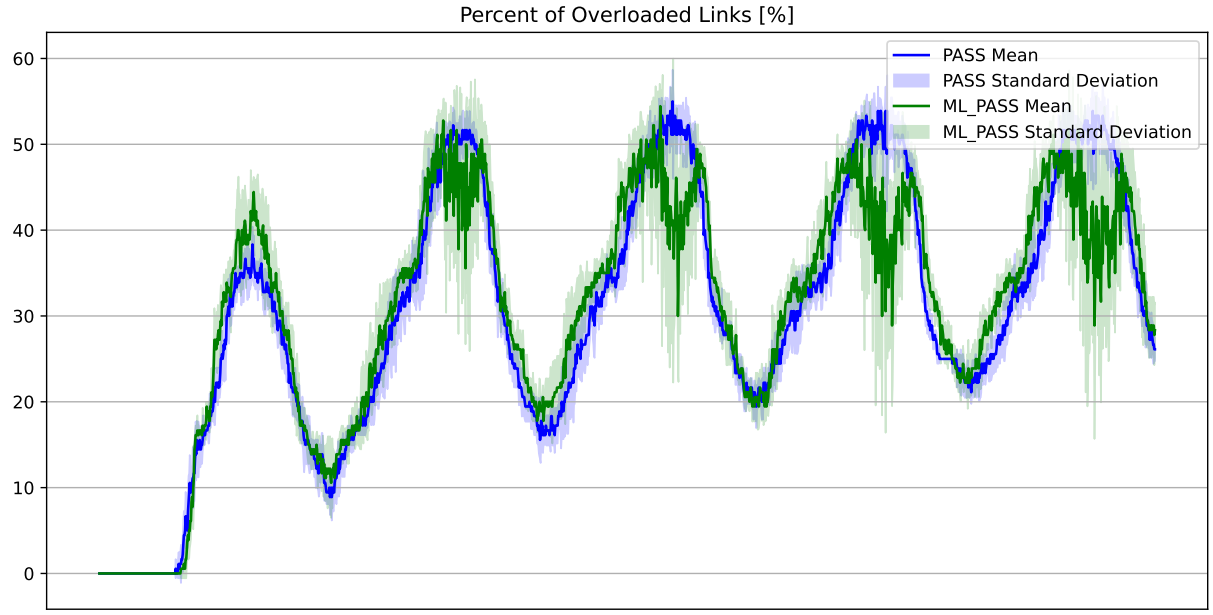

Figure 6.52: Overloaded links for *PASS* and *210* generator rate.

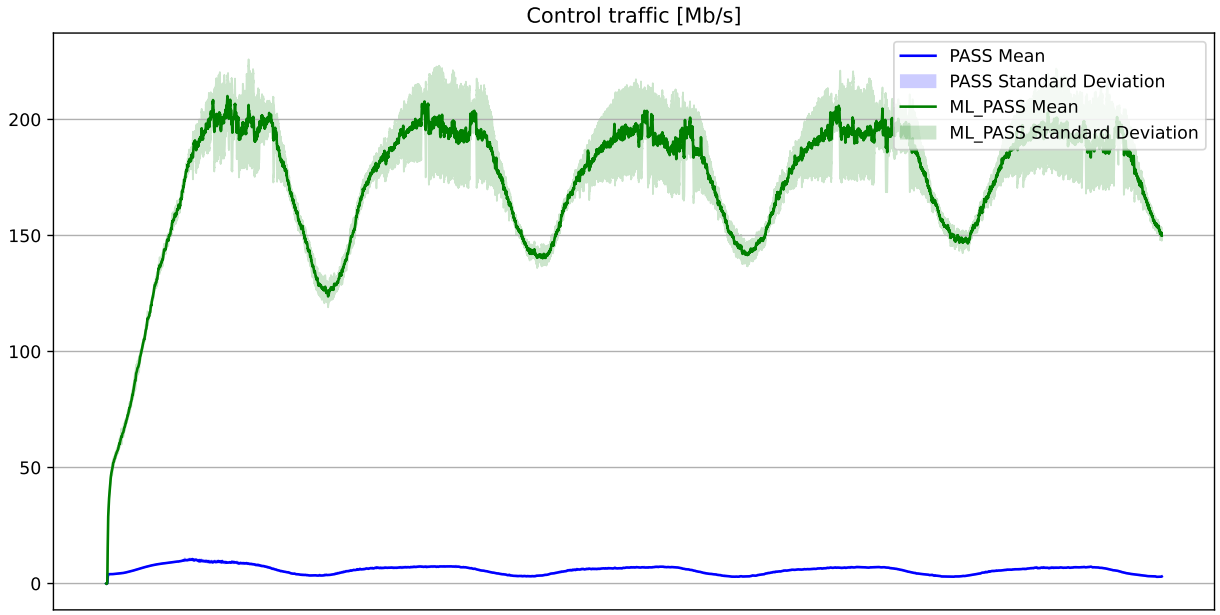

Figure 6.53: Control traffic for *PASS* and *210* generator rate.

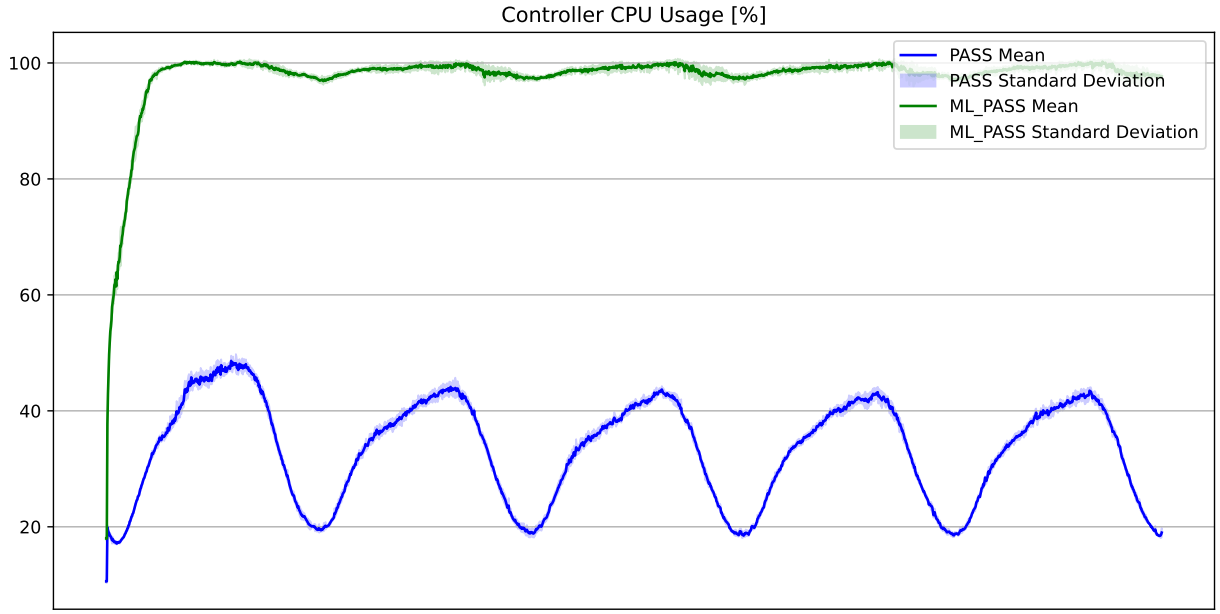

Figure 6.54: Controller CPU usage for *PASS* and *210* generator rate.

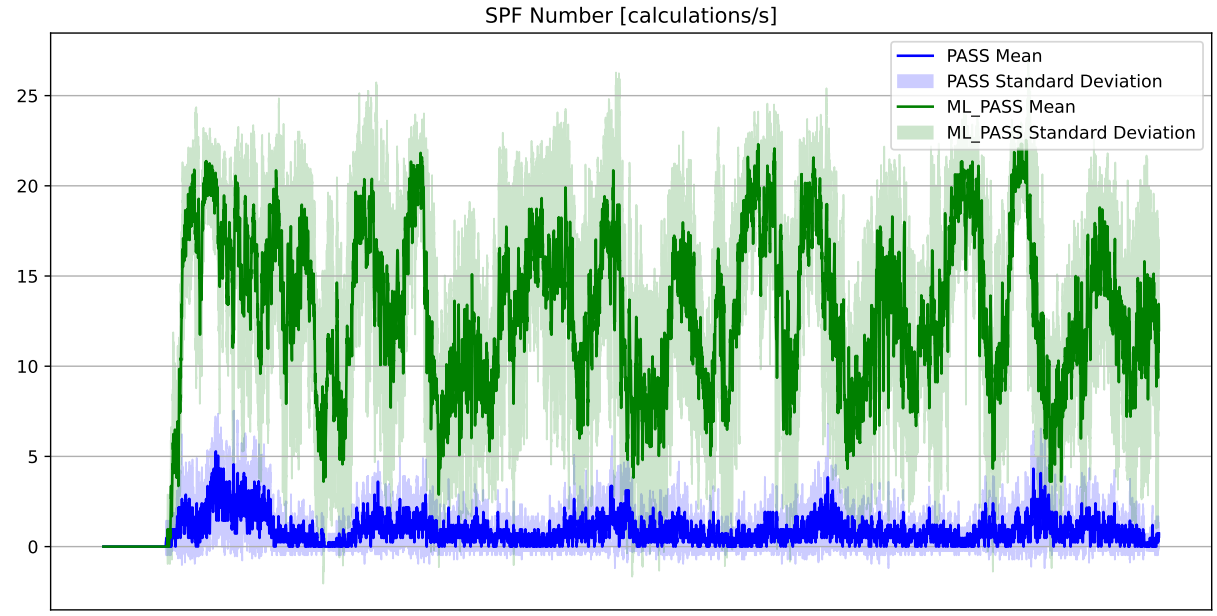

Figure 6.55: SPF calculations for *PASS* and *210* generator rate.

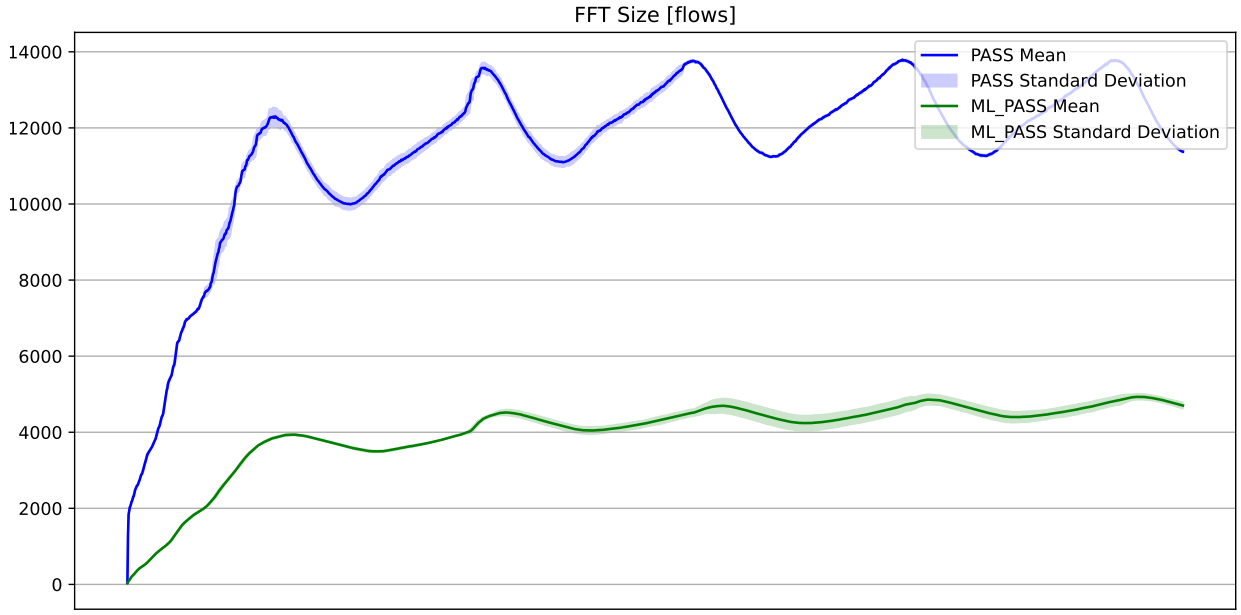

Figure 6.56: FFT size for *PASS* and *210* generator rate.

6.3 Discussion

The results for the classic reactive mode, *NO_PASS*, reveal several limitations and distinctive patterns that manifest across all simulated generator rates, ranging from a saturated network state to an over-saturated state. The analysis of link statistics reveals that the link throughput is consistently lower in *ML_NO_PASS* compared to *NO_PASS*. This observation indicates a substantial reduction in the total transmitted traffic under the *ML_NO_PASS* configuration. Such a reduction is evident across all generator rates (150, 180, 210, and 240), as illustrated in Figures 6.5, 6.23, 6.41, and 6.59. A direct outcome of this reduced link throughput is a notable decrease in link delay, which can be attributed to diminished instances of link overload. This inverse relationship between link throughput and delay is consistently observed across all generator rates (150, 180, 210, and 240), as demonstrated in Figures 6.6, 6.24, 6.42, and 6.60. In addition to the reduction in delay, the lower volume of transmitted traffic results in a decrease in the number of overloaded links. This trend is confirmed by the results presented in Figures 6.7, 6.25, 6.43, and 6.61. From the perspective of controller performance, a marked increase in control traffic is noted for all generator rates. As the generator rate escalates, a proportional rise in control traffic is also evident, as depicted in Figures 6.8, 6.26, 6.44, and 6.62. Consequently, the controller's CPU usage experiences a significant surge in *ML_NO_PASS*, which is expected due to the additional computational demands associated with inferences, as shown in Figures 6.9, 6.27, 6.45, and 6.63. The reduction in transmitted traffic is directly correlated with a decline in the number of SPF calculations. This trend is consistently observed across all generator rates, as shown in Figures 6.10, 6.28, 6.46, and 6.64. Furthermore, a substantial reduction is seen in both the FFT size and the number of FFT operations, particularly the FFT addition operations. This reduction in size and operations under *ML_NO_PASS* is evident for all generator rates, as demonstrated in Figures 6.11, 6.29, 6.47, 6.65, 6.12, 6.30, 6.48, and 6.66. Lastly, for lower generator rates (150 and 180), the observed losses remain relatively constant, as illustrated in Figures 6.13 and 6.31. However, at higher generator rates (210 and 240), the daily traffic envelope of the generator becomes increasingly pronounced, as depicted in Figures 6.49 and 6.67. The persistently high loss rate in *ML_NO_PASS*, coupled with lower link throughputs, increased link delays, and reduced SPF calculations, is directly correlated. **In *ML_NO_PASS*, the heightened loss rate is primarily a consequence of packets being discarded when a flow is not classified as an elephant. As described earlier, this operational characteristic of *NO_PASS* mode results in the discarding of non-elephant flow packets, thereby contributing to elevated loss rates.**

In contrast to the *NO_PASS* operation mode, the results for the *PASS* mode demonstrate significantly improved network performance. This improvement stems from the intended behavior of the *PASS* mode, where flows classified as mouse flows are effectively forwarded using the existing routing tables and the SPF algorithm. This way of operation allowed to keep the link throughput almost perfectly aligned (with a small advantage for the *ML_PASS*) between the *ML_PASS* and *PASS* operation modes for all generator rates as shown in Figures 6.14, 6.32, 6.50, and 6.68. Link delays for lower generator rates (150, and 180) are a little bit higher for the *ML_PASS*, however lower for the higher generator rates (210, and 240) as shown in Figures 6.15, 6.33, 6.51, and 6.69. The same pattern is visible for the number of overloaded links as illustrated in Figures 6.16, 6.34, 6.52, and 6.70. The increased control traffic is expected, it is also increasing together with the generator rate increase as shown in Figures 6.17, 6.35, 6.53, and 6.71. It is also

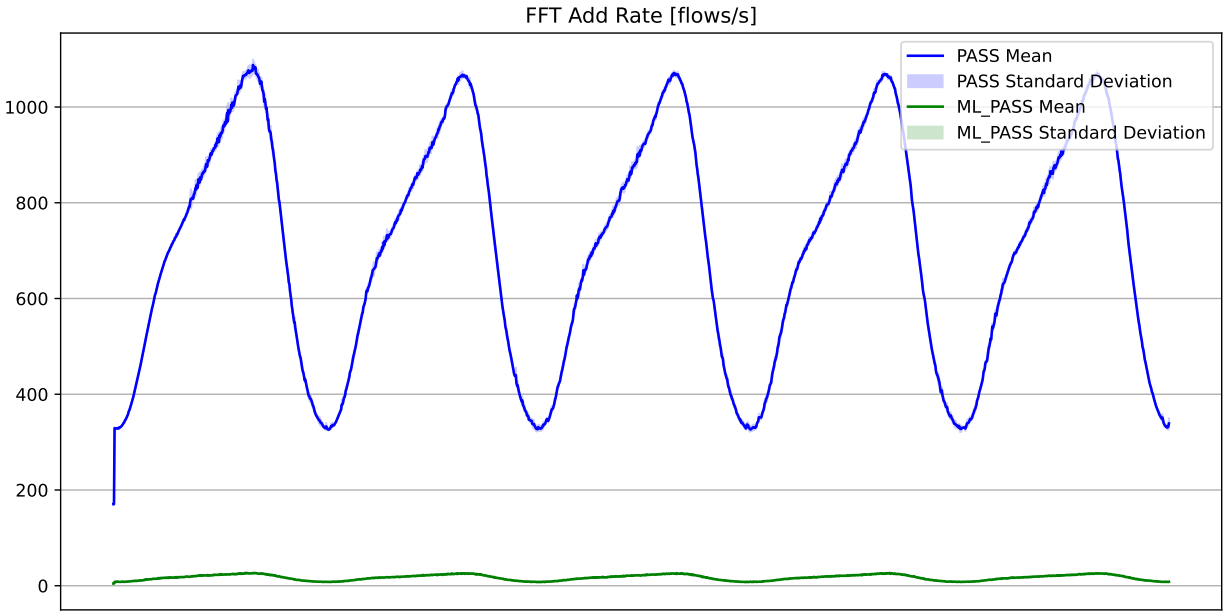

Figure 6.57: FFT add operations for *PASS* and *210* generator rate.

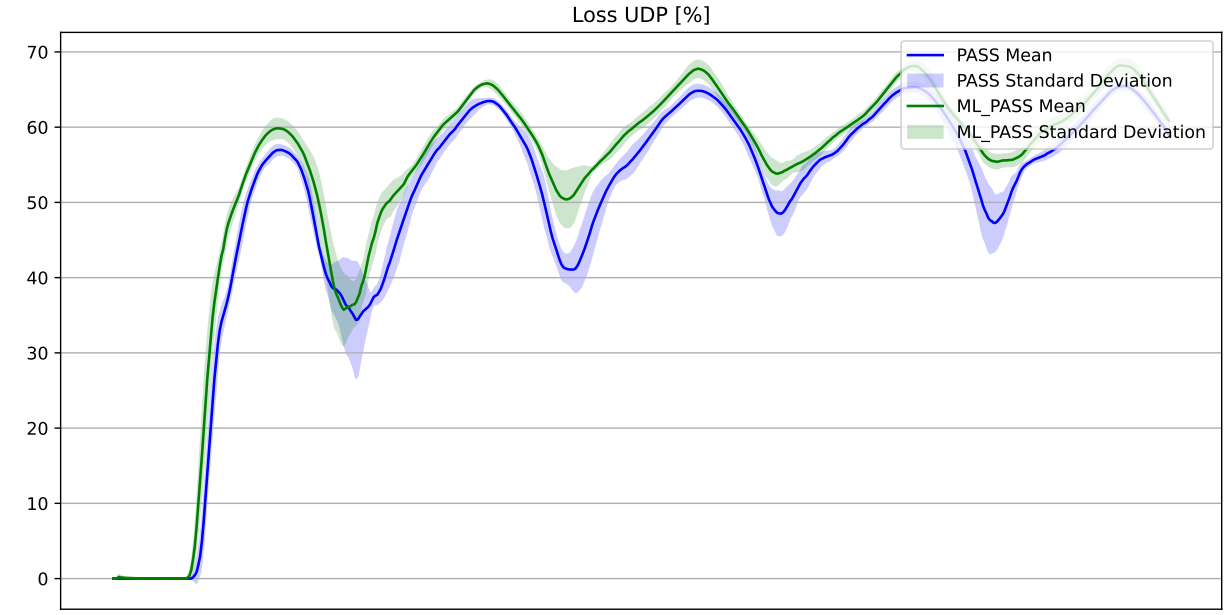

Figure 6.58: UDP losses for *PASS* and *210* generator rate.

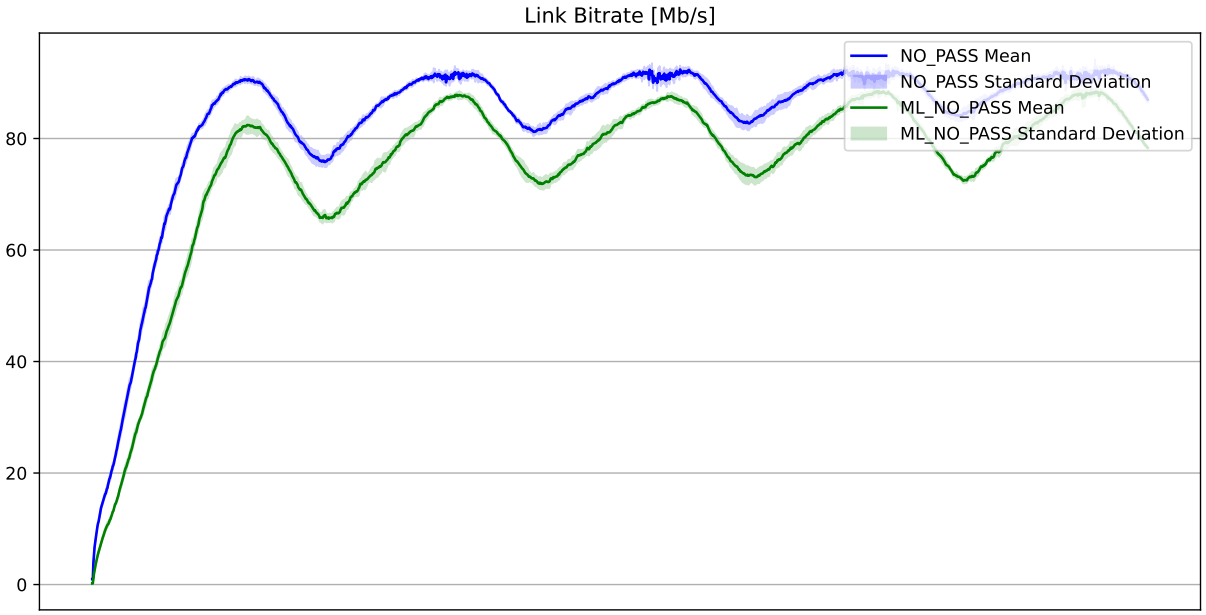

Figure 6.59: Link throughput for *NO_PASS* and *240* generator rate.

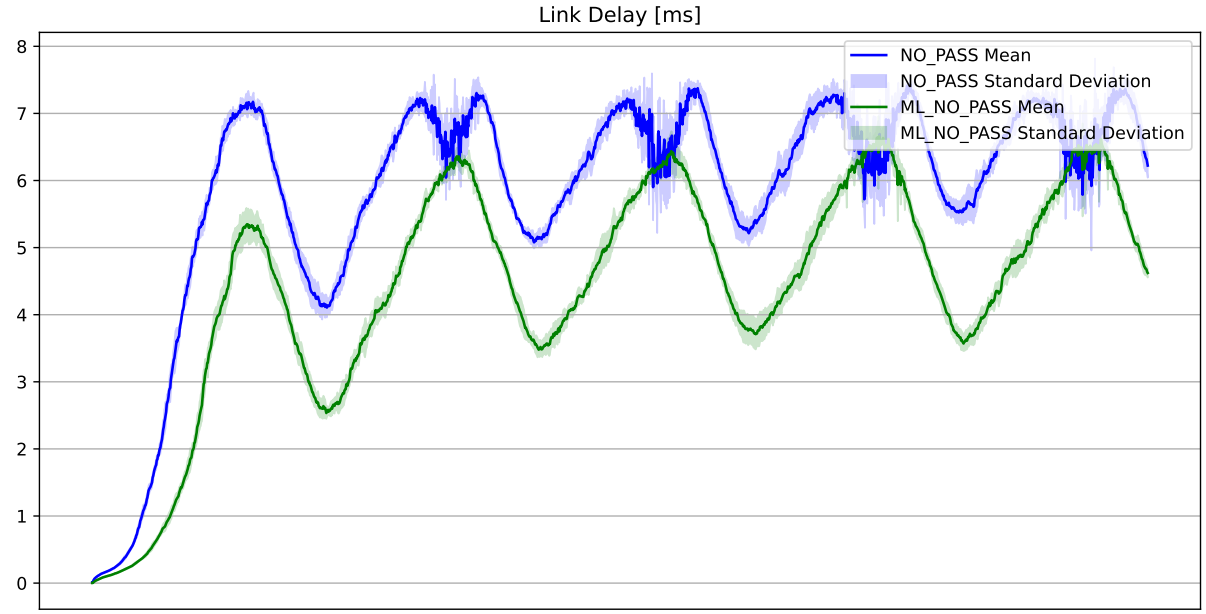

Figure 6.60: Link delay for *NO_PASS* and *240* generator rate.

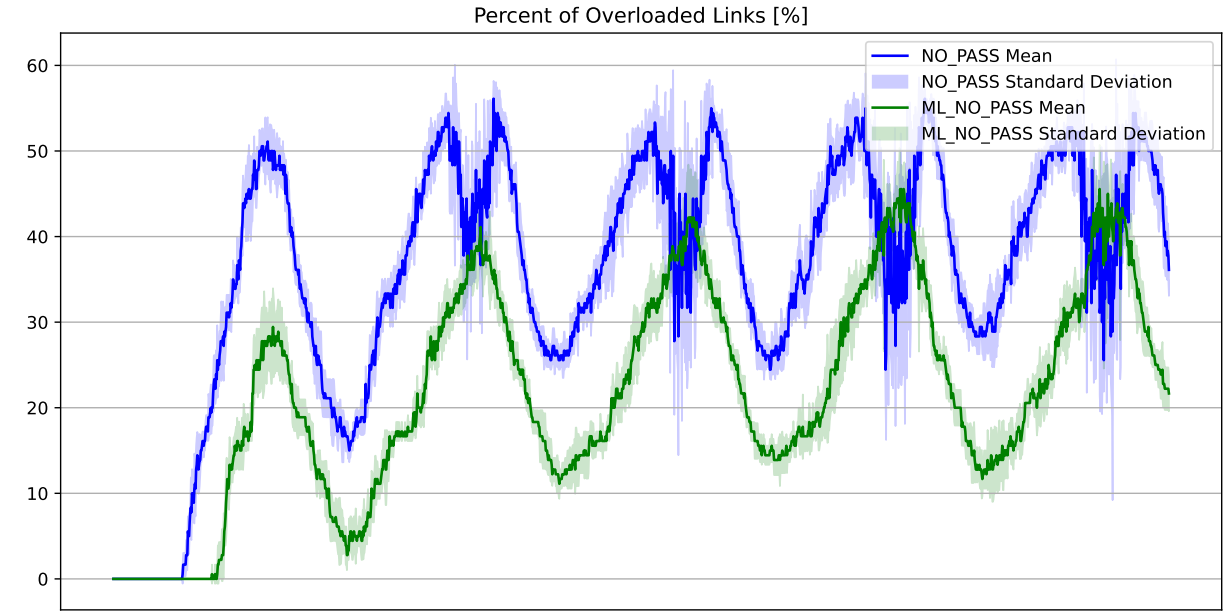

Figure 6.61: Overloaded links for *NO_PASS* and *240* generator rate.

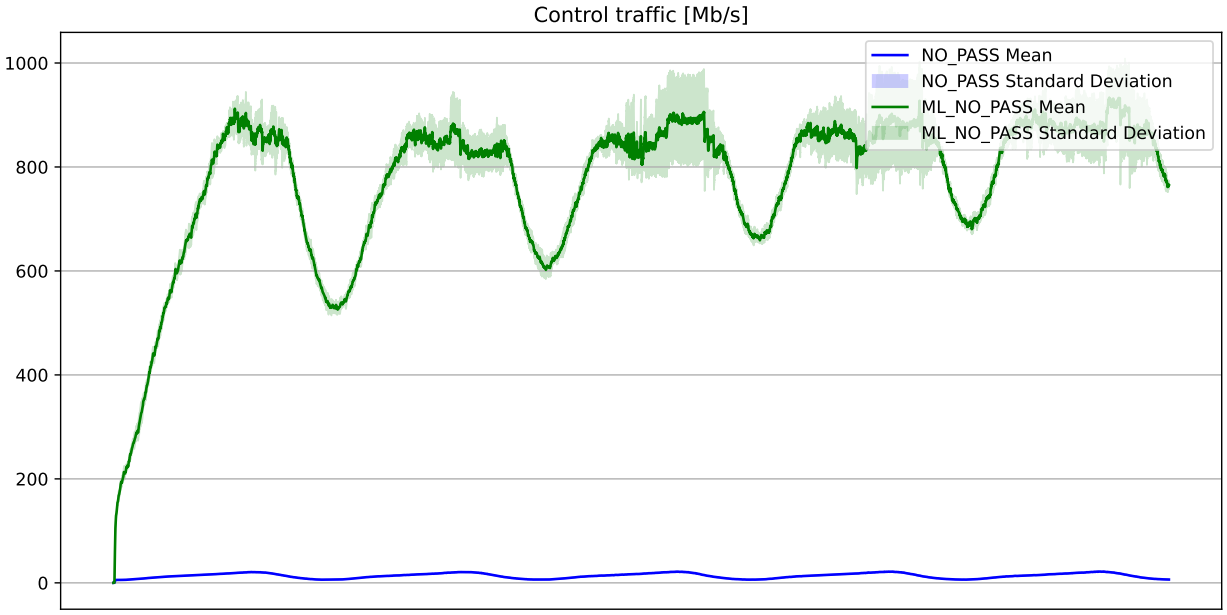

Figure 6.62: Control traffic for *NO_PASS* and *240* generator rate.

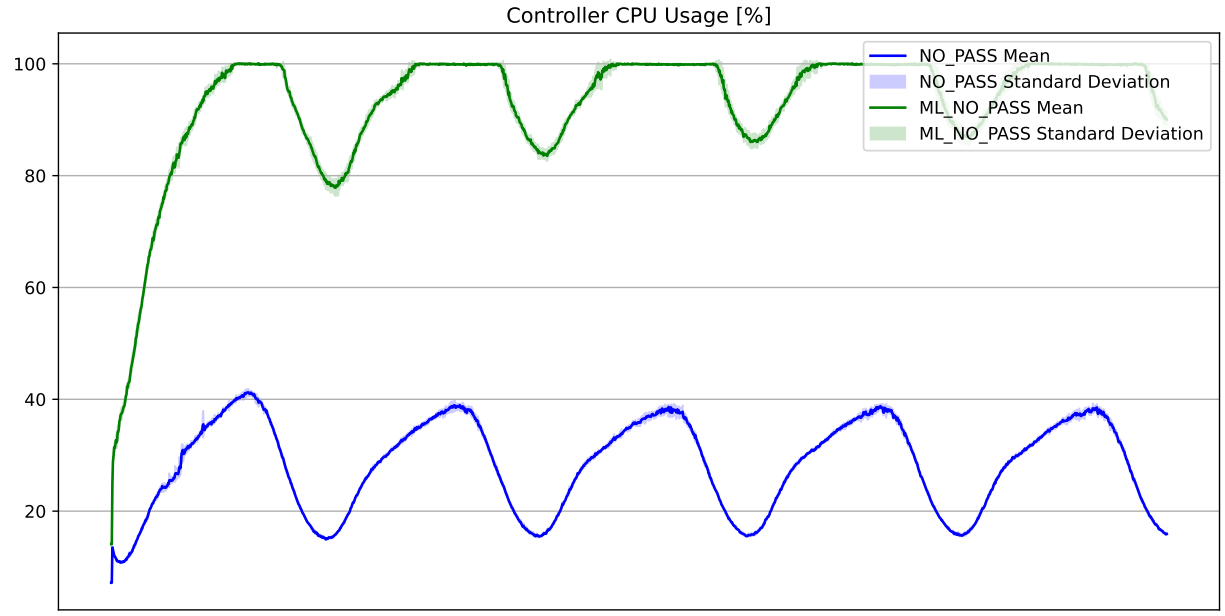

Figure 6.63: Controller CPU usage for *NO_PASS* and *240* generator rate.

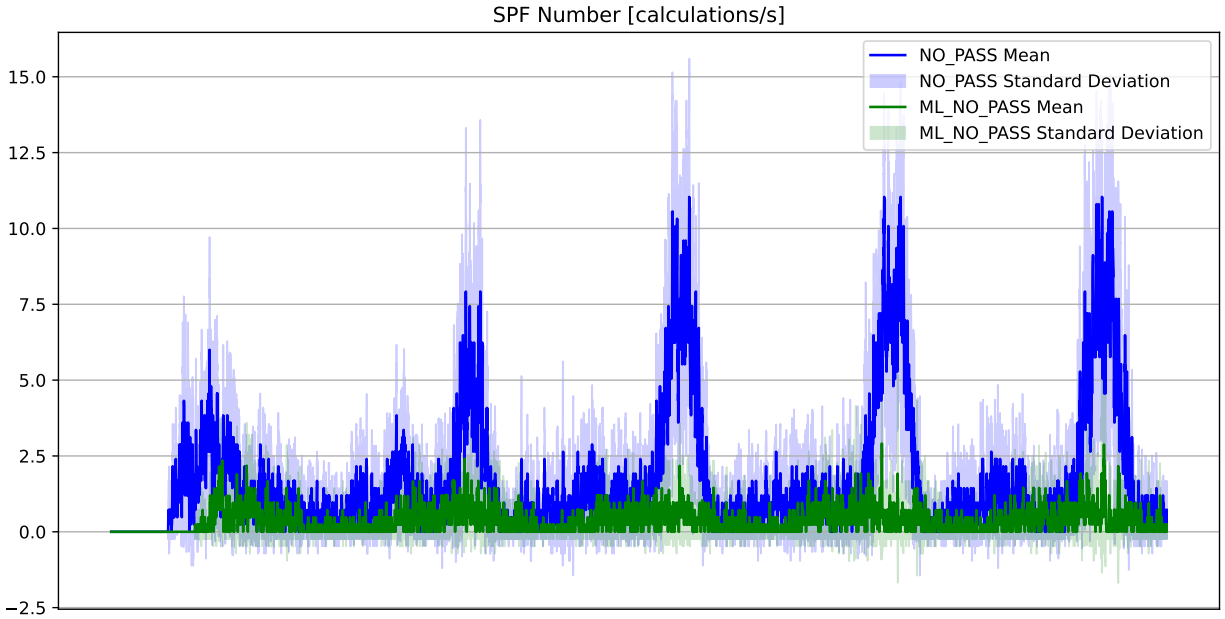

Figure 6.64: SPF calculations for *NO_PASS* and *240* generator rate.

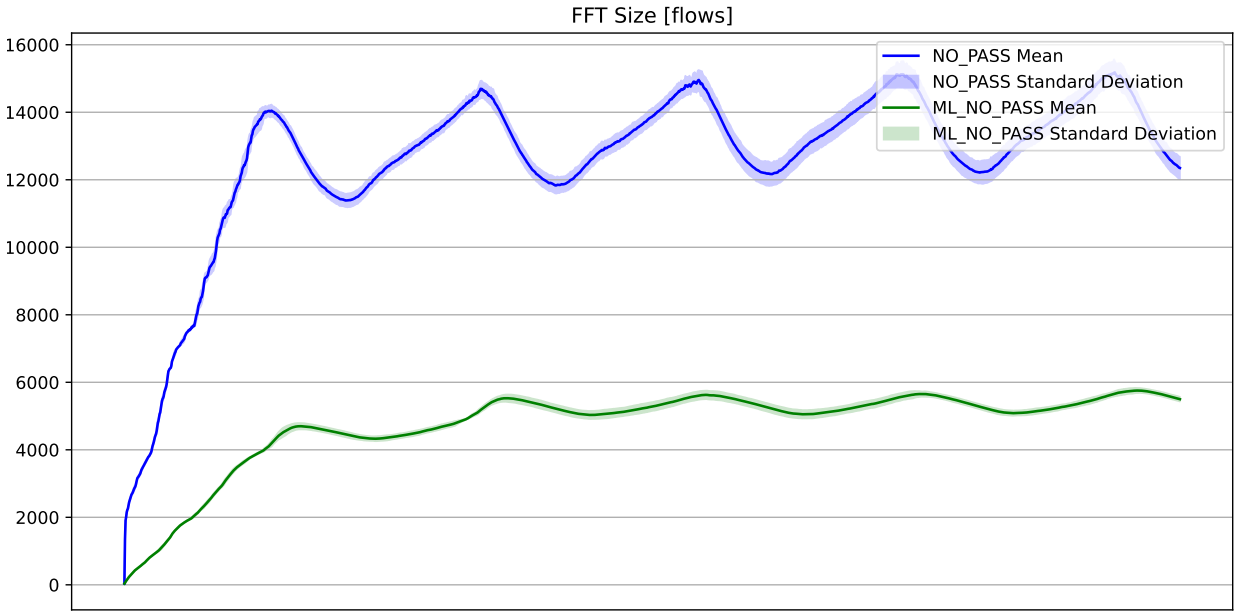

Figure 6.65: FFT size for *NO_PASS* and *240* generator rate.

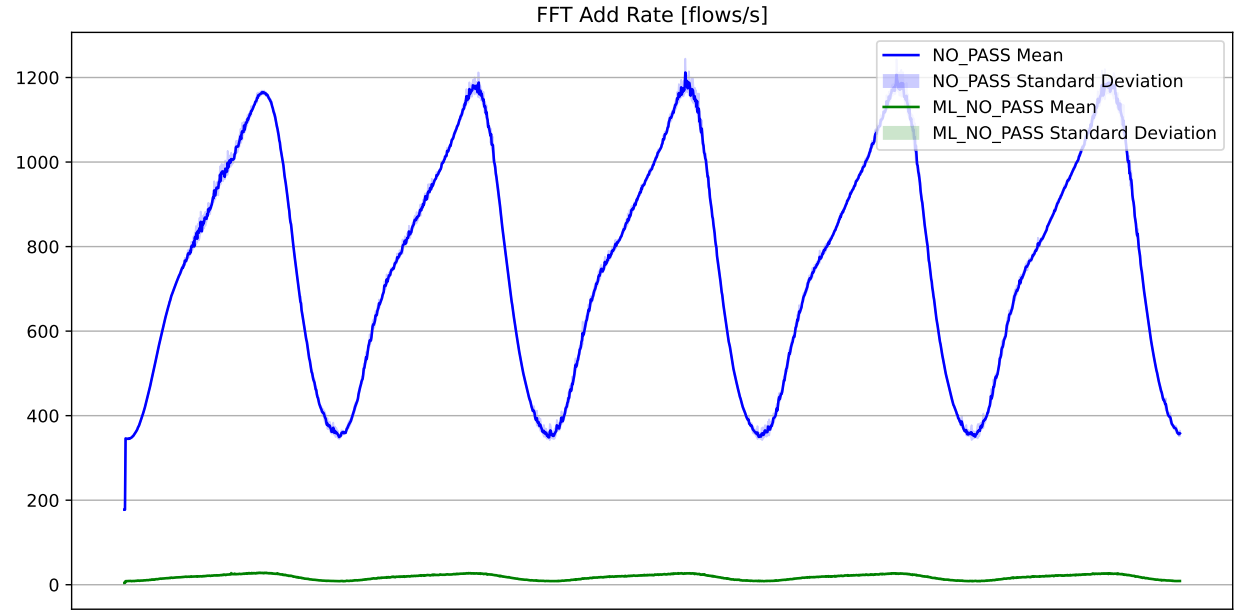

Figure 6.66: FFT add operations for *NO_PASS* and *240* generator rate.

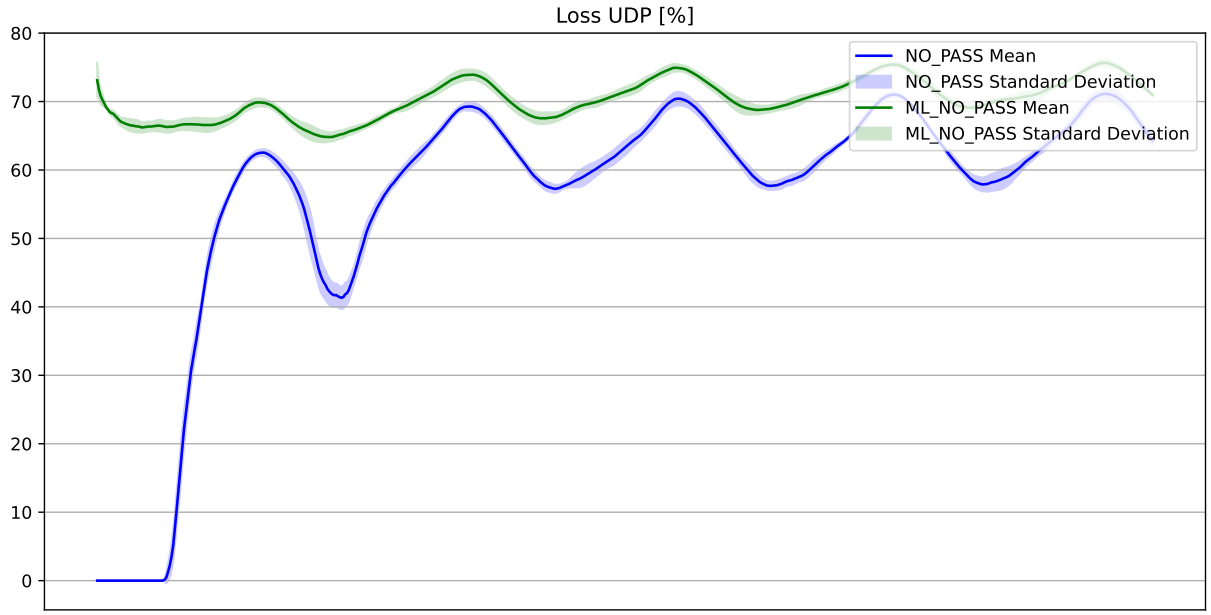

Figure 6.67: UDP losses for *NO_PASS* and *240* generator rate.

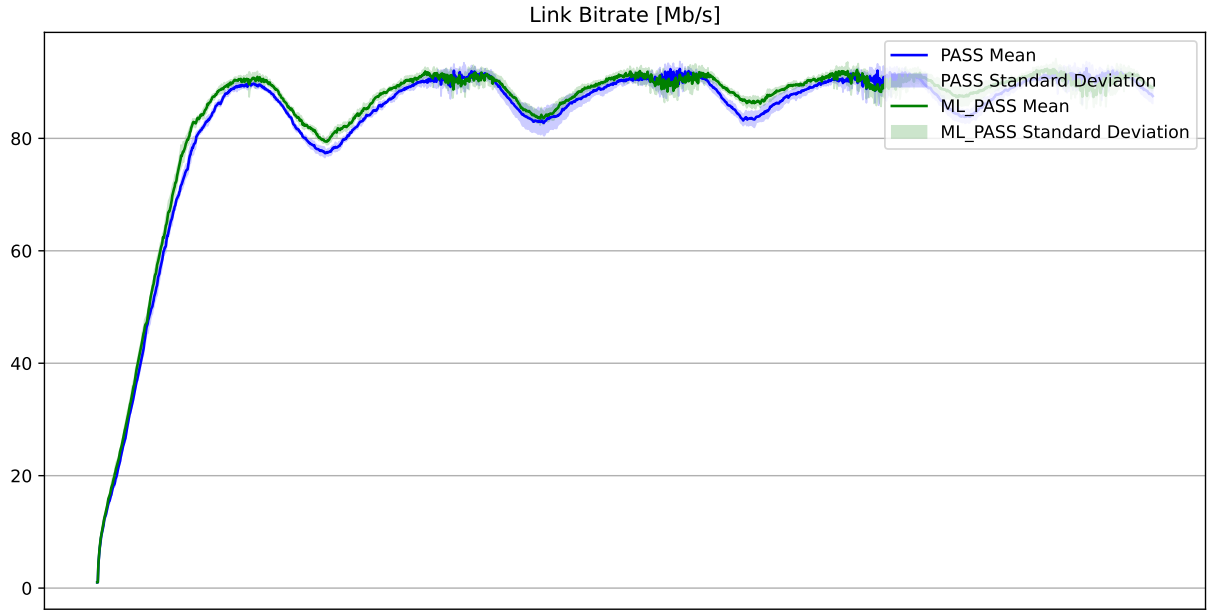

Figure 6.68: Link throughput for *PASS* and *240* generator rate.

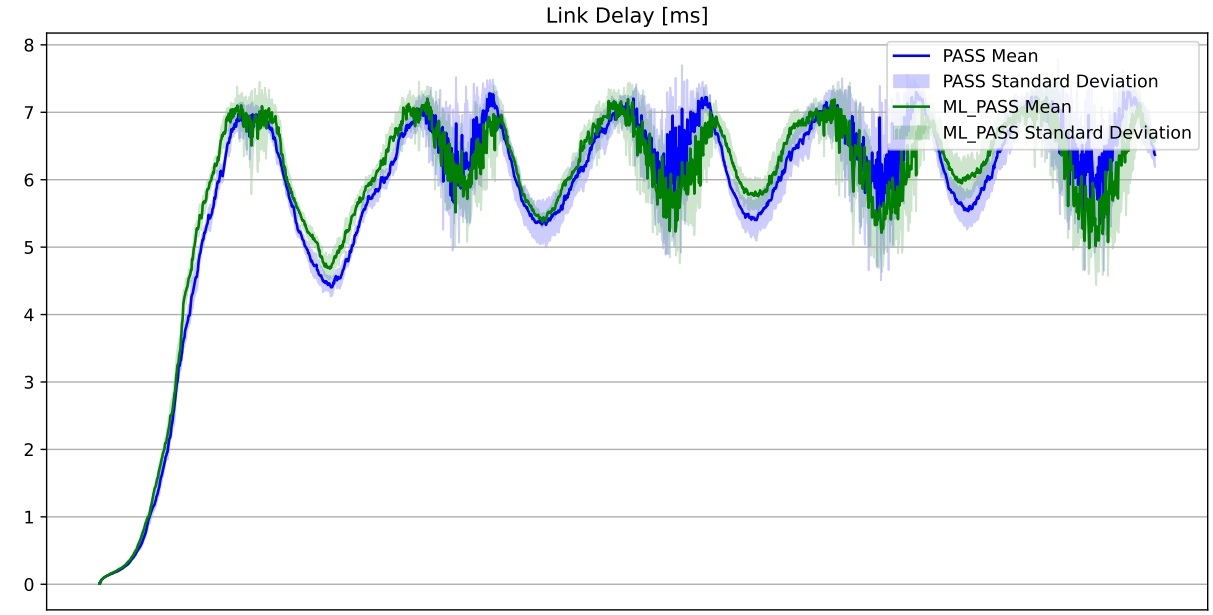

Figure 6.69: Link delay for *PASS* and *240* generator rate.

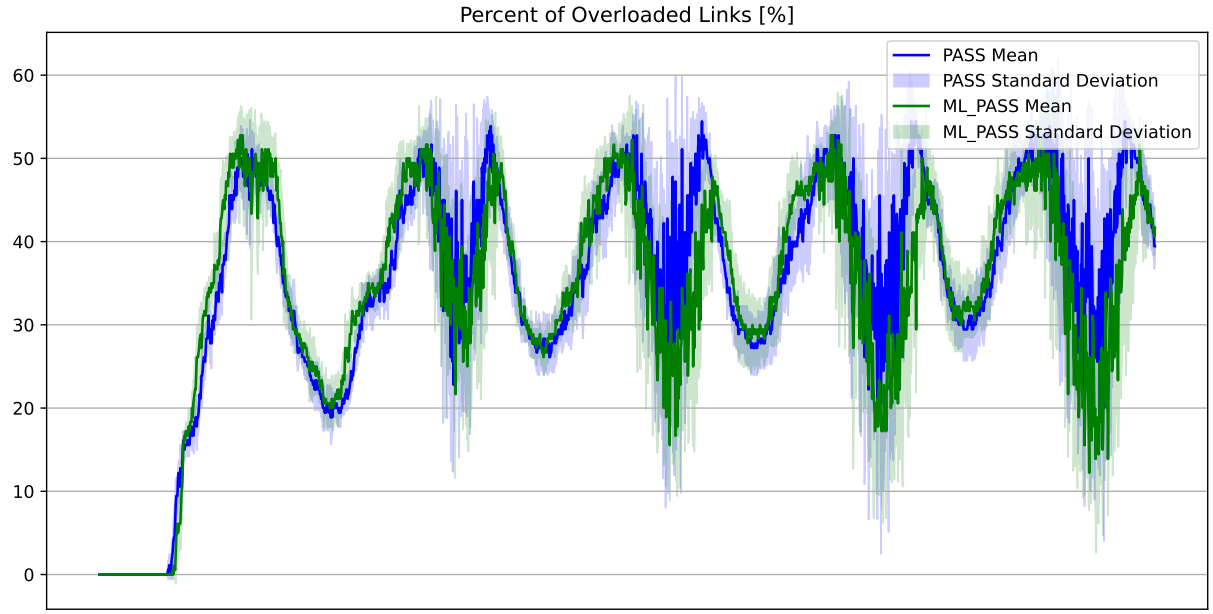

Figure 6.70: Overloaded links for *PASS* and *240* generator rate.

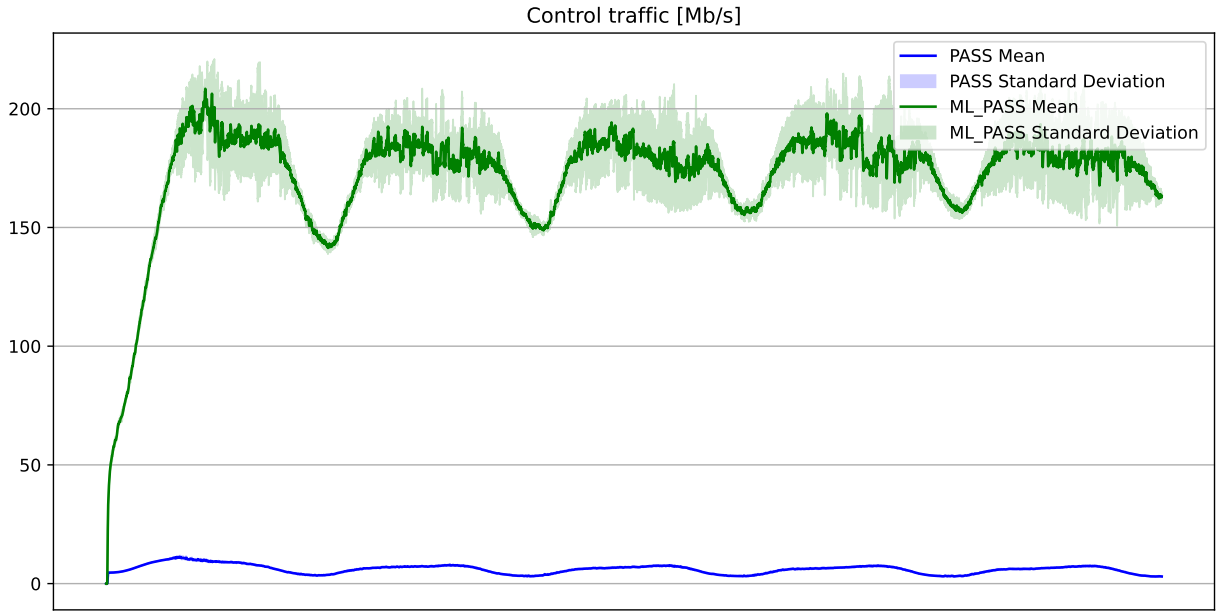

Figure 6.71: Control traffic for *PASS* and *240* generator rate.

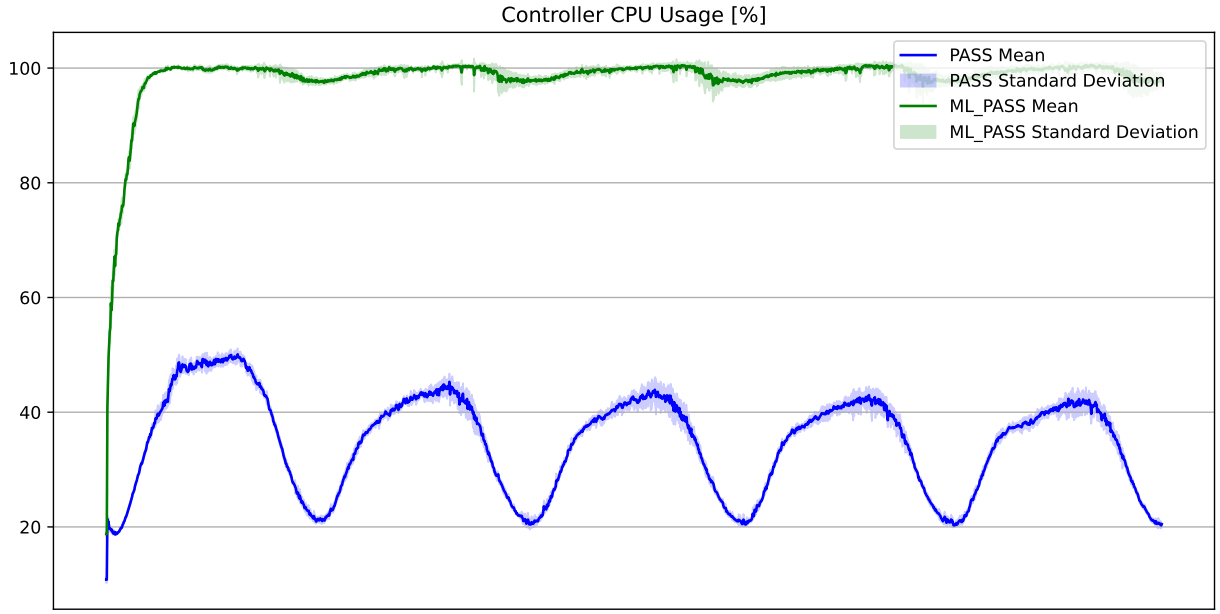

Figure 6.72: Controller CPU usage for *PASS* and *240* generator rate.

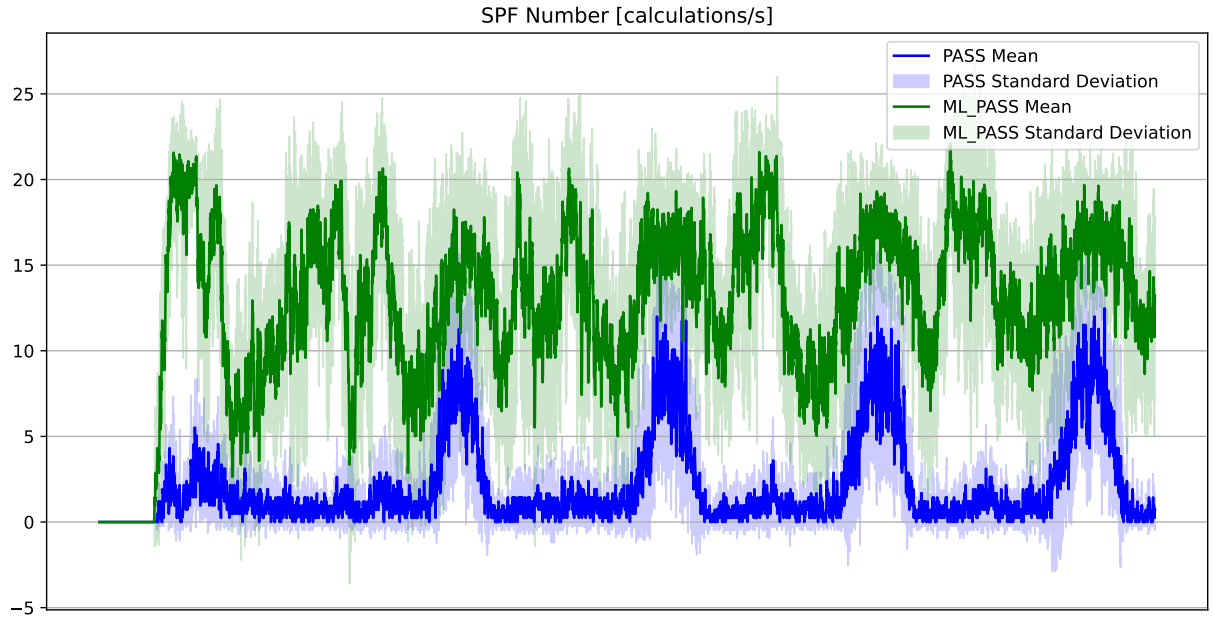

Figure 6.73: SPF calculations for *PASS* and *240* generator rate.

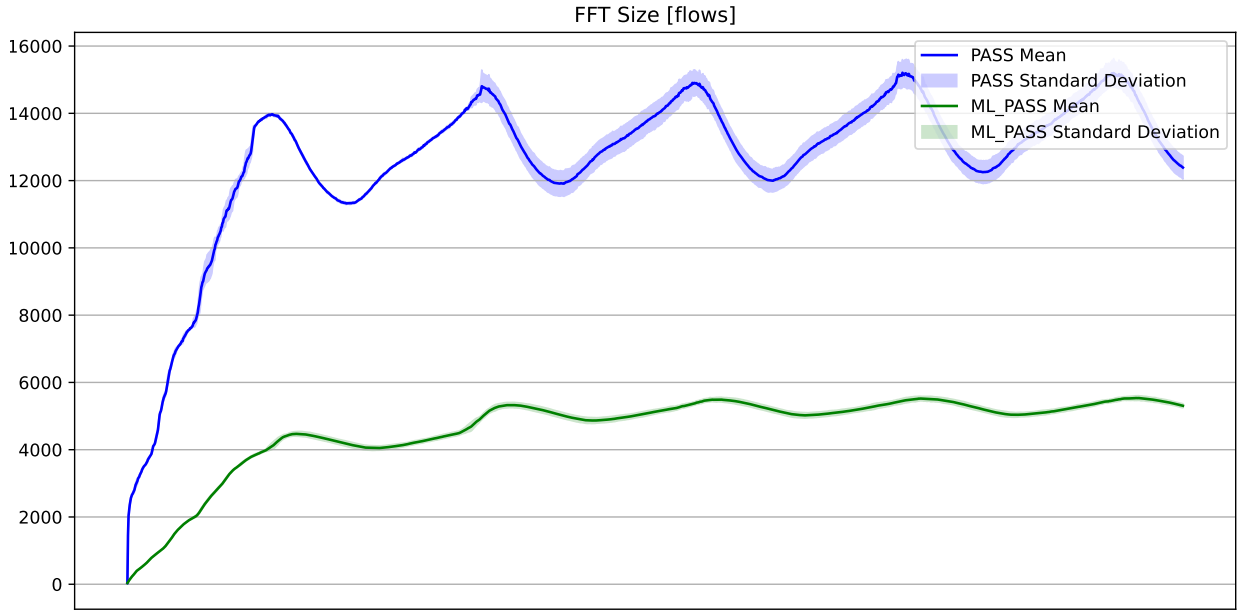

Figure 6.74: FFT size for *PASS* and *240* generator rate.

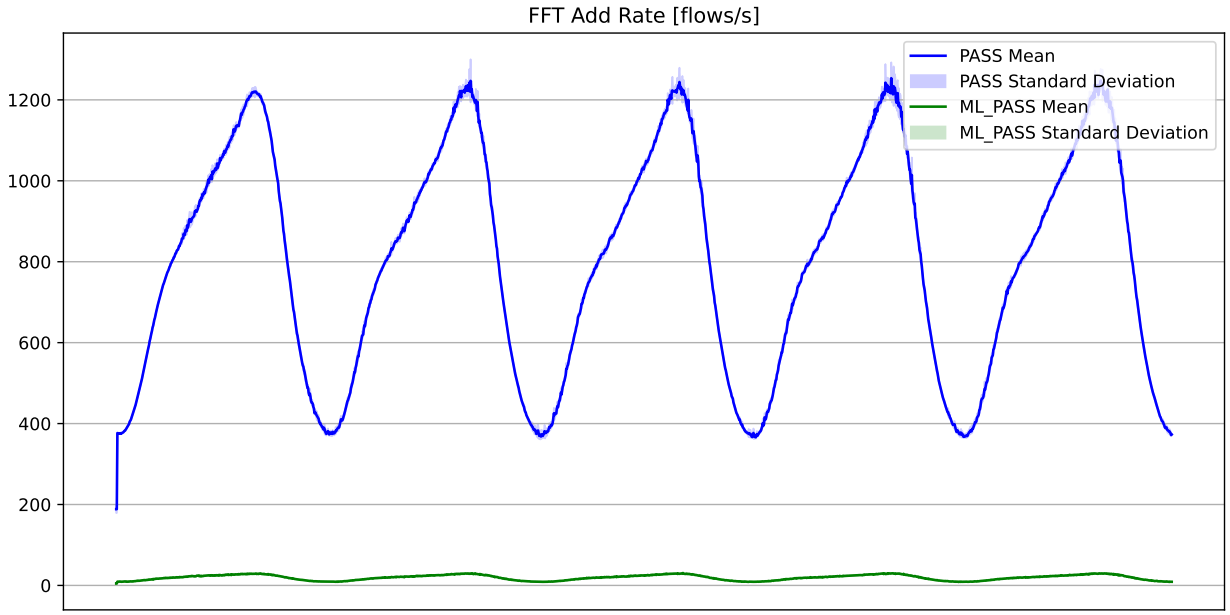

Figure 6.75: FFT add operations for *PASS* and *240* generator rate.

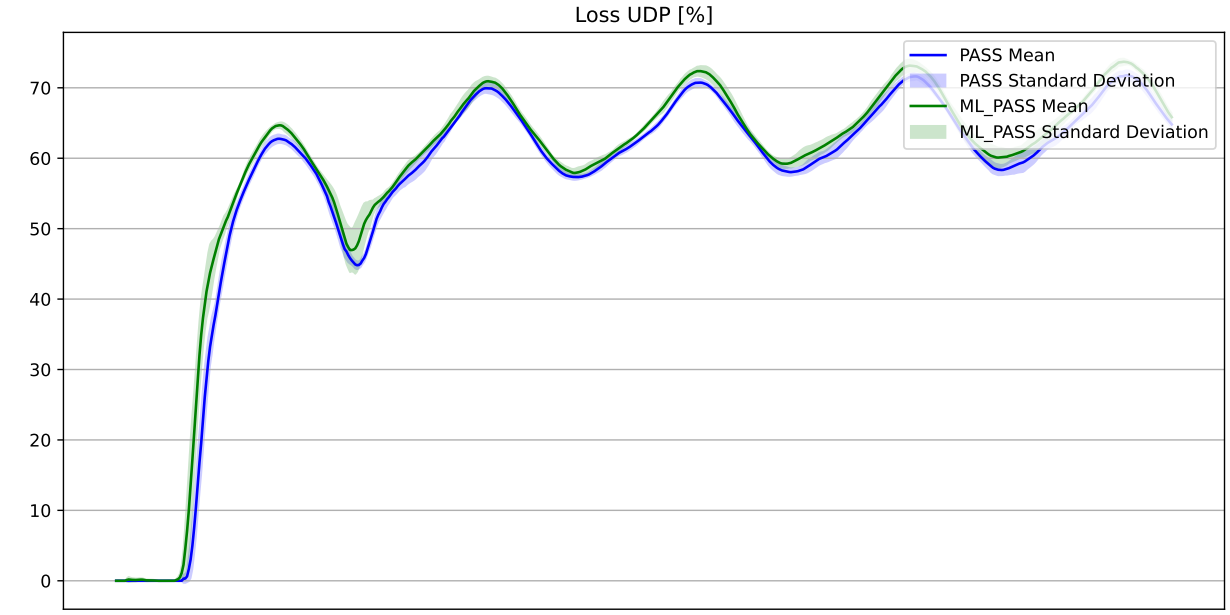

Figure 6.76: UDP losses for *PASS* and *240* generator rate.

expected that control traffic is much lower in *ML_PASS* than in the *ML_NO_PASS* since in *ML_PASS* only the packet's header is sent to the controller. Controller's CPU usage is expectedly higher for the *ML_PASS* than for the operation mode without the ML for all generator rates visible in Figures 6.18, 6.36, 6.54, and 6.72. It is also clearly visible that CPU utilization for the *PASS* operation mode is significantly higher than for the *NO_PASS*. The direct cause of this is the increase in the number of the SPF calculations which also is expected in the *PASS* mode. The increase in the number of the SPF calculations is observed for all generator rates as depicted in Figures 6.19, 6.37, 6.55, and 6.73. Similarly, as in the *ML_NO_PASS*, *ML_PASS* provides a substantial reduction in both the FFT size and the number of FFT addition operations. This reduction is evident for all generator rates, as demonstrated in Figures 6.11, 6.29, 6.47, 6.65, 6.12, 6.30, 6.48, and 6.66. Finally, regarding the UDP losses in the network, the pattern is the same as for the link delay, and overload percentage. The losses are higher for lower generator rates (150, and 180). Meanwhile, for higher generator rates (210, and 240) the losses are almost perfectly aligned with the *PASS* operation mode. This pattern is shown in Figures 6.22, 6.40, 6.58, and 6.76. The initial overhead of classification in *ML_PASS* is more noticeable at lower traffic levels. However, as traffic increases, the benefits of more intelligent and adaptive routing outweigh this initial overhead, leading to improved network characteristics at higher generator rates.

The forwarding strategy in the *PASS* mode improves network performance by leveraging intelligent flow classification. The *ML_PASS* mode effectively reduces the number of overloaded links, which further contributes to maintaining network stability and preventing bottlenecks. However, the introduction of machine learning-based classification in *ML_PASS* significantly increases the controller's CPU usage. Increased CPU usage of the controller is a cost of significant reduction of the FFT operations and FFT size in the network switches.

In summary, the *ML_PASS*, enhances network performance by intelligently segregating flow types. This approach effectively reduces network congestion, optimizes resource utilization, and ensures a higher QoS for end-users, particularly in scenarios with higher traffic loads.

Chapter 7

Finale

The goal of the research presented in this dissertation was to prove that it is possible to improve network traffic management and enhance QoS by integrating SDN, ML-based flow classification, and FAMTAR, with a specific focus on early elephant flow identification and handling. Throughout the research, various mechanisms and strategies were proposed, implemented, and evaluated to validate this hypothesis.

The research focused on two main areas: flow classification and traffic engineering. In the area of flow classification, the goal was to identify large network flows, known as *elephant flows*, right from the flow start. This early classification is crucial for optimizing routing and resource allocation. The research demonstrated that using information solely from the 5-tuple allowed for accurate flow classification. To achieve this, various normalization methods, input data representations, and problem formulations were explored. The findings highlighted that representing input data in terms of *bits* outperformed other representations in all evaluated models.

The effectiveness of the models was thoroughly assessed, with the best results summarized in Table 5.11. The models varied in complexity and effectiveness in reducing flow operations, with the best results obtained using the *ExtraTreesClassifier* from the *scikit-learn* library, which achieved a reduction of 66.35% in flow operations. The exploration and comparison of different machine learning models led to a clear understanding of the trade-offs between accuracy, computational cost, and memory usage.

In the context of traffic engineering, the dissertation introduced the traffic engineering system which incorporates machine learning-based classification into traditional routing mechanisms. The system demonstrates a significant improvement in network performance by effectively segregating flow types and optimizing routing strategies. By intelligently routing elephant and mouse flows, the presented approach manages to reduce network congestion and improve resource utilization. However, the introduction of machine learning classification comes with the cost of increased controller CPU usage, but is beneficial in reducing the number and size of FFT operations within network switches, ultimately leading to a more stable and efficient network.

7.1 Achievements and contributions

The achievements and contributions of the dissertation can be summarized as follows:

- **Accurate and Early Flow Classification**: The study confirmed that it is possible to classify flows accurately at their start using only information from the 5-tuple. The findings showed that using more comprehensive input representations, such as *bits*, offered superior accuracy at the expense of increased computational costs.
- **Comparison of Machine Learning Models**: The dissertation presented a thorough evaluation of multiple machine learning models for flow classification, exploring different normalization methods, input data representations, and problem formulations. This comparative analysis offers valuable insights into model selection and optimization for future researchers and network engineers looking to implement similar solutions.
- **Integration of Machine Learning with FAMTAR and SDN**: The dissertation contributes a novel integration of machine learning with FAMTAR and SDN. This hybrid approach combines the intelligence of machine learning with the efficiency of established network protocols, setting the stage for more adaptive and resilient network designs.
- **Machine Learning for Network Traffic Management**: Integrating machine learning models into the routing decision process provided clear benefits, especially in improving traffic flow distribution and reducing network congestion. Although this required higher computational power in the controller, it significantly optimized network performance by reducing FFT operations in switches.
- **Optimized Traffic Engineering**: The traffic engineering system, enhanced by machine learning, demonstrated its effectiveness in improving QoS for users. By dynamically identifying and handling elephant flows, the presented approach enabled a more balanced and efficient use of network resources, ensuring stable and reliable performance, especially under heavy traffic loads.
- **Reduction of Flow Table Size and Operations**: By accurately classifying flows and intelligently routing them, the research successfully reduced the size and number of operations in the FFT. This reduction in FFT operations results in more efficient memory usage and faster packet processing within network switches. This is a crucial finding as it addresses a common limitation in SDN architectures where limited flow table capacity and processing speed can become bottlenecks.

In light of the presented achievements it can be stated that the thesis:

It is possible to improve network traffic management and enhance Quality of Service by integrating Software-Defined Networking, Machine Learning, and Flow-Aware Multi-Topology Adaptive Routing, with a specific focus on early elephant flow identification and handling.

has been proved.

Bibliography

[1] Nauman Ahad, Junaid Qadir, and Nasir Ahsan. Neural networks in wireless networks: Techniques, applications and guidelines. *Journal of Network and Computer Applications*, 68:1–27, 2016.

[2] Michal Aibin. Traffic prediction based on machine learning for elastic optical networks. *Optical Switching and Networking*, 30:33–39, 2018.

[3] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks*, 71:1–30, 2014.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 19, USA, 2010. USENIX Association.

[5] Nahlah Abdulrahman Alkhalidi and Fouad A Yaseen. Fdphi: Fast deep packet header inspection for data traffic classification and management. *International Journal of Intelligent Engineering & Systems*, 14(4), 2021.

[6] Kelvin Santos Amorim and Gustavo Sousa Pavani. Ant colony optimization-based distributed multilayer routing and restoration in ip/mpls over optical networks. *Computer Networks*, 185:107747, 2021.

[7] Sercan Ö Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 6679–6687, 2021.

[8] Ola Ashour, Marc St-Hilaire, Thomas Kunz, and Maoyu Wang. A survey of applying reinforcement learning techniques to multicast routing. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 1145–1151, 2019.

[9] Ahmad Azab, Mahmoud Khasawneh, Saed Alrabaee, Kim-Kwang Raymond Choo, and Maysa Sarsour. Network traffic classification: Techniques, datasets, and challenges. *Digital Communications and Networks*, 10(3):676–692, 2024.

[10] Abdelhadi Azzouni and Guy Pujolle. Neutm: A neural network-based framework for traffic matrix prediction in sdn. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5, 2018.

[11] Dario Bega, Marco Gramaglia, Albert Banchs, Vincenzo Sciancalepore, and Xavier Costa-Perez. A machine learning approach to 5g infrastructure market optimization. *IEEE Transactions on Mobile Computing*, 19(3):498–512, 2020.

[12] Gérard Biau and Erwan Scornet. A random forest guided tour. *Test*, 25:197–227, 2016.

[13] Piotr Borylo, Edyta Biernacka, Jerzy Domzal, Bartosz Kadziolka, Miroslaw Kantor, Krzysztof Rusek, Maciej Skala, Krzysztof Wajda, Robert Wojcik, and Wojciech Zabek. A tutorial on reinforcement learning in selected aspects of communications and networking. *Computer Communications*, 208:89–110, 2023.

[14] Piotr Borylo, Edyta Biernacka, Jerzy Domzal, Bartosz Kadziolka, Miroslaw Kantor, Krzysztof Rusek, Maciej Skala, Krzysztof Wajda, Robert Wojcik, and Wojciech Zabek. Neural networks in selected aspects of communications and networking. *IEEE Access*, 2024.

[15] Piotr Boryło, Piotr Chołda, Jerzy Domżał, Piotr Jaglarz, Piotr Jurkiewicz, Michał Rzepka, Grzegorz Rzym, and Robert Wójcik. Sdnroute: Proactive routing optimization in software defined networks. *Computer Communications*, 225:250–278, 2024.

[16] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, Jun 2018.

[17] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[18] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[19] Marcus Vinicius Brito da Silva, Alberto Egon Schaeffer-Filho, and Lisandro Zambenedetti Granville. HashCuckoo: Predicting Elephant Flows using Meta-Heuristics in Programmable Data Planes. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 6337–6342, 2022.

[20] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019.

[21] P. Castoldi, A. Giorgetti, A. Sgambelluri, F. Paolucci, and F. Cugini. Segment routing in multi-layer networks. In *2017 19th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4, July 2017.

[22] Tania Cerquitelli, Michela Meo, Marilia Curado, Lea Skorin-Kapov, and Eirini Eleni Tsiropoulou. Machine learning empowered computer networks, 2023.

[23] M. Chen, U. Challita, W. Saad, C. Yin, and M. Debbah. Artificial neural networks-based machine learning for wireless networks: A tutorial. *IEEE Communications Surveys Tutorials*, 21(4):3039–3071, 2019.

[24] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[25] CISCO. Cisco Annual Internet Report (2018–2023) White Paper. 2020.

[26] Bruno L. Coelho and Alberto E. Schaeffer-Filho. CrossBal: Data and Control Plane Cooperation for Efficient and Scalable Network Load Balancing. In *2023 19th International Conference on Network and Service Management (CNSM)*, pages 1–9, 2023.

[27] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[28] Marcus Vinicius Brito da Silva, Arthur Selle Jacobs, Ricardo José Pfitscher, and Lisandro Zambenedetti Granville. Predicting Elephant Flows in Internet Exchange Point Programmable Networks. In *International Conference on Advanced Information Networking and Applications*, pages 485–497. Springer, 2019.

[29] R. Durner and W. Kellerer. Network Function Offloading Through Classification of Elephant Flows. *IEEE Transactions on Network and Service Management*, 17(2):807–820, June 2020.

[30] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[31] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[32] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[33] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[34] B. Gangopadhyay, J. Santos, J. Pedro, J. Ukkonen, M. Hannula, J. Kemppainen, J. Nieminen, P. Ylisirnio, V. Hallivuori, M. Silvola, A. Kankkunen, T. Doiron, J. Baldry, N.R. Prasad, and R. Prasad. Sdn-enabled slicing in disaggregated multi-domain and multi-layer 5g transport networks. In *2020 23rd International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 1–6, Oct 2020.

[35] Yisel Gari, David A. Monge, Elina Pacini, Cristian Mateos, and Carlos García Garino. Reinforcement learning-based application autoscaling in the cloud: A survey. *Engineering Applications of Artificial Intelligence*, 102:104288, 2021.

[36] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63:3–42, 2006.

[37] Alessio Giorgetti, Andrea Sgambelluri, Francesco Paolucci, Filippo Cugini, and Piero Castoldi. Demonstration of dynamic restoration in segment routing multi-layer sdn networks. In *2016 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2016.

[38] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM computer communication review*, 38(3):105–110, 2008.

[39] Otkrist Gupta and Ramesh Raskar. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

[40] Jorge Gómez, Velssy Hernandez Riaño, and Gustavo Ramirez-Gonzalez. Traffic classification in ip networks through machine learning techniques in final systems. *IEEE Access*, 11:44932–44940, 2023.

[41] Mosab Hamdan, Bushra Mohammed, Usman Humayun, Ahmed Abdelaziz, Suleman Khan, M Akhtar Ali, Muhammad Imran, and Muhammad Nadzir Marsono. Flow-aware elephant flow detection for software-defined networks. *IEEE Access*, 8:72585–72597, 2020.

[42] C. Hardegen, B. Pfülb, S. Rieger, and A. Gepperth. Predicting Network Flow Characteristics Using Deep Learning and Real-World Network Traffic. *IEEE Transactions on Network and Service Management*, 17(4):2662–2676, 2020.

[43] Christoph Hardegen, Benedikt Pfülb, Sebastian Rieger, Alexander Gepperth, and Sven Reißmann. Flow-based throughput prediction using deep learning and real-world network traffic. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–9, 2019.

[44] M.H. Hassoun and A.P.C.E.M.H. Hassoun. *Fundamentals of Artificial Neural Networks*. A Bradford book. MIT Press, 1995.

[45] A. He, K. K. Bae, T. R. Newman, J. Gaeddert, K. Kim, R. Menon, L. Morales-Tirado, J. '. Neel, Y. Zhao, J. H. Reed, and W. H. Tranter. A survey of artificial intelligence for cognitive radios. *IEEE Transactions on Vehicular Technology*, 59(4):1578–1592, 2010.

[46] Heng He, Zhezhe Peng, Xiaohu Zhou, and Jia Wang. Lfod: A lightweight flow table optimization scheme in sdn based on flow length distribution in the internet. In *2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–6, 2022.

[47] Hani Jamjoom, Dan Williams, and Upendra Sharma. Don't call them middleboxes, call them middlepipes. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 19–24, 2014.

[48] Nathan Jay, Noga H. Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. *International Conference on Machine Learning*, pages 3050–3059, 2019.

[49] P. Jurkiewicz. Boundaries of Flow Table Usage Reduction Algorithms Based on Elephant Flow Detection. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2021.

[50] Piotr Jurkiewicz. flow-models 2.0: Elephant flows modeling and detection with machine learning. *SoftwareX*, 24:101506, 2023.

[51] Piotr Jurkiewicz, Bartosz Kadziołka, Mirosław Kantor, Jerzy Domżał, and Robert Wójcik. Machine learning-based elephant flow classification on the first packet. *IEEE Access*, 12:105744–105760, 2024.

[52] Piotr Jurkiewicz, Bartosz Kądziołka, Mirosław Kantor, Robert Wójcik, and Jerzy Domżał. Elephant flow detection with random forest models under programmable network dataplane constraints. *IEEE Access*, pages 1–1, 2024.

[53] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. Flow length and size distributions in campus Internet traffic. *Computer Communications*, 167:15–30, 2021.

[54] Piotr Jurkiewicz, Robert Wójcik, Jerzy Domżał, and Andrzej Kamisiński. Testing implementation of FAMTAR: Adaptive multipath routing. *Computer Communications*, 149:300–311, 2020.

[55] Bartosz Kądziołka and Piotr Jurkiewicz. Redukcja wielkości tablicy przepływów dzięki detekcji dużych przepływów za pomocą uczenia maszynowego. *Przegląd Telekomunikacyjny+ Wiadomości Telekomunikacyjne*, 2023.

[56] Bartosz Kądziołka, Piotr Jurkiewicz, Robert Wójcik, and Jerzy Domżał. Utilizing tabnet deep learning for elephant flow detection by analyzing information in first packet headers. *Entropy*, 26(7):537, 2024.

[57] Bartosz Kądziołka, Maciej Skała, Robert WóJCIK, Piotr Jurkiewicz, and Jerzy Domżał. Employing famtar and ahb to achieve an optical resource-efficient multilayer ip-over-eon sdn network. *IEEE Access*, 10:94089–94099, 2022.

[58] Mirosław Kantor, Edyta Biernacka, Piotr Boryło, Jerzy Domżał, Piotr Jurkiewicz, Miłosz Stypiński, and Robert Wójcik. A survey on multi-layer IP and optical Software-Defined Networks. *Computer Networks*, 162:106844, 2019.

[59] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[60] Nikhil Ketkar, Jojo Moolayil, Nikhil Ketkar, and Jojo Moolayil. Introduction to pytorch. *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*, pages 27–91, 2021.

[61] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[62] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv e-prints*, pages arXiv–1412, 2014.

[63] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M.Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, pages 263–297, August 2000.

[64] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, Jan. 2015.

[65] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[66] Bartosz Kądziołka, Piotr Jurkiewicz, Robert Wójcik, and Jerzy Domżał. Elephant flow classification on the first packet with neural networks. *IEEE Access*, 12:65298–65309, 2024.

[67] Jong Hyouk Lee and Kamal Singh. Switchtree: in-network computing and traffic analyses with random forests. *Neural Computing and Applications*, 2020.

[68] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2019.

[69] S. Lin, I. F. Akyildiz, P. Wang, and M. Luo. Qos-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 25–33, June 2016.

[70] Siqi Liu, Wei Lu, and Zuqing Zhu. On the cross-layer orchestration to address ip router outages with cost-efficient multilayer restoration in ip-over-eons. *Journal of Optical Communications and Networking*, 10:A122–A132, 2018.

[71] Wai-Xi Liu, Jun Cai, Yu Wang, Qing Chun Chen, and Jia-Qi Zeng. Fine-grained flow classification using deep learning for software defined data center networks. *Journal of Network and Computer Applications*, 168:102766, 2020.

[72] Yong Liu, Tianyi Yu, Qian Meng, and Quanze Liu. Flow optimization strategies in data center networks: A survey. *Journal of Network and Computer Applications*, 226:103883, 2024.

[73] Manuel Lopez-Martin, Belen Carro, Jaime Lloret, Santiago Egea, and Antonio Sanchez-Esguevillas. Deep Learning Model for Multimedia Quality of Experience Prediction Based on Network Flow Packets. *IEEE Communications Magazine*, 56(9):110–117, 2018.

[74] Meilian Lu, Yun Gu, and Dongliang Xie. A dynamic and collaborative multi-layer virtual network embedding algorithm in sdn based on reinforcement learning. *IEEE Transactions on Network and Service Management*, 17(4):2305–2317, 2020.

[75] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[76] Nick McKeown. How sdn will shape networking. *Open Networking Summit*, 2011.

[77] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[78] Péter Megyesi and Sándor Molnár. Analysis of elephant users in broadband network traffic. In *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*, pages 37–45. Springer, 2013.

[79] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero. A Survey on the Contributions of Software-Defined Networking to Traffic Engineering. *IEEE Communications Surveys Tutorials*, 19(2):918–953, 2017.

[80] Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. Memory-efficient random forests in fpga smartnics. CoNEXT 2023, page 55–56, New York, NY, USA, 2023. Association for Computing Machinery.

[81] James N Morgan and John A Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American statistical association*, 58(302):415–434, 1963.

[82] Suyel Namasudra, Pascal Lorenz, and Uttam Ghosh. The new era of computer network by using machine learning. *Mobile Networks and Applications*, 28(2):764–766, 2023.

[83] Peter Newman, Greg Minshall, and Thomas L Lyon. Ip switching-atm under ip. *IEEE/ACM Transactions on networking*, 6(2):117–129, 1998.

[84] P. Jurkiewicz. flow-models: A framework for analysis and modeling of IP network flows. *SoftwareX*, 17:100929, 2022.

[85] Francesco Paolucci, Filippo Cugini, and Piero Castoldi. P4-based multi-layer traffic engineering encompassing cyber security. In *Optical Fiber Communication Conference*, pages M4A–5. Optical Society of America, 2018.

[86] Panos Papanikolaou, Konstantinos Christodoulopoulos, and Emmanouel Varvarigos. Joint multi-layer survivability techniques for ip-over-elastic-optical-networks. *Journal of Optical Communications and Networking*, 9(1):A85–A98, 2017.

[87] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[88] Adrian Pekar, Alejandra Duque-Torres, Winston K.G. Seah, and Oscar M. Caicedo Rendon. Towards threshold-agnostic heavy-hitter classification. *International Journal of Network Management*, 32(3):e2188, 2022.

[89] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[90] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Yanhui Geng, Li Chen, K. Chen, and Hao Jin. Online flow size prediction for improved network routing. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–6, 2016.

[91] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. Online flow size prediction for improved network routing. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–6, 2016.

[92] Yichen Qian, Jun Wu, Rui Wang, Fusheng Zhu, and Wei Zhang. Survey on reinforcement learning applications in communication networks. *Journal of Communications and Information Networks*, 4(2):30–39, 2019.

[93] Zhaorong Qian, Guoju Gao, and Yang Du. Per-flow size measurement by combining sketch and flow table in software-defined networks. In *2022 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 644–651, 2022.

[94] Chao Qiu, Shaohua Cui, Haipeng Yao, Fangmin Xu, F. Richard Yu, and Chenglin Zhao. A novel QoS-enabled load scheduling algorithm based on reinforcement learning in software-defined energy internet. *Future Generation Computer Systems*, 92:43–51, 2019.

[95] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[96] Nesma M. Rezk, Madhura Purnaprajna, Tomas Nordström, and Zain Ul-Abdin. Recurrent neural networks: An embedded computing perspective. *IEEE Access*, 8:57967–57996, 2020.

[97] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[98] Ćiril Rožić, Dimitrios Klonidis, and Ioannis Tomkos. A survey of multi-layer network optimization. In *2016 International Conference on Optical Network Design and Modeling (ONDM)*, pages 1–6. IEEE, 2016.

[99] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[100] Berta Serracanta, Alberto Rodriguez-Natal, Fabio Maino, and Albert Cabellos. Flow optimization at inter-datacenter networks for application run-time acceleration, 2024.

[101] A. Sgambelluri, A. Giorgetti, F. Cugini, G. Bruno, F. Lazzeri, and P. Castoldi. First demonstration of sdn-based segment routing in multi-layer networks. In *2015 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3, March 2015.

[102] Anees Shaikh, Jennifer Rexford, and Kang G. Shin. Load-sensitive Routing of Long-lived IP Flows. *ACM SIGCOMM Computer Communication Review*, 29(4):215–226, 1999.

[103] Gengbiao Shen, Qing Li, Shuo Ai, Yong Jiang, Mingwei Xu, and Xuya Jia. How powerful switches should be deployed: A precise estimation based on queuing theory. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 811–819. IEEE, 2019.

[104] Xiaoming Tao, Yiping Duan, Mai Xu, Zhishen Meng, and Jianhua Lu. Learning QoE of Mobile Video Transmission with Deep Neural Network: A Data-Driven Approach. *IEEE Journal on Selected Areas in Communications*, 37(6):1337–1348, 2019.

[105] Aashma Uprety and Danda B. Rawat. Reinforcement learning for iot security: A comprehensive survey. *IEEE Internet of Things Journal*, 8:8693–8706, 2021.

[106] Shie-Yuan Wang and Ying-Hua Wu. Supporting large random forests in the pipelines of a hardware switch to classify packets at 100-gbps line rate. *IEEE Access*, 11:112384–112397, 2023.

[107] Wenbo Wang, Andres Kwasinski, Dusit Niyato, and Zhu Han. A survey on applications of model-free strategy learning in cognitive wireless networks. *IEEE Communications Surveys Tutorials*, 18(3):1717–1757, 2016.

[108] Getahun Wassie, Jianguo Ding, and Yihenew Wondie. Traffic prediction in sdn for explainable qos using deep learning approach. *Scientific Reports*, 13(1), 2023.

[109] Robert Wójcik and Andrzej Jajszczyk. Flow oriented approaches to qos assurance. *ACM Computing Surveys (CSUR)*, 44(1):1–37, 2012.

[110] Robert Wójcik, Jerzy Domżal, and Zbigniew Duliński. Flow-Aware Multi-Topology Adaptive Routing. *IEEE Communications Letters*, 18(9):1539–1542, Sep 2014.

[111] Peng Xiao, Wenyu Qu, Heng Qi, Yujie Xu, and Zhiyang Li. An efficient elephant flow detection with cost-sensitive in sdn. In *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pages 24–28, 2015.

[112] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu. A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys Tutorials*, 21(1):393–430, 2019.

[113] Shengxu Xie, Guyu Hu, Changyou Xing, and Yaqun Liu. Online elephant flow prediction for load balancing in programmable switch-based dcn. *IEEE Transactions on Network and Service Management*, 21(1):745–758, 2024.

[114] Hongli Xu, He Huang, Shigang Chen, and Gongming Zhao. Scalable software-defined networking through hybrid switching. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[115] Fouad A Yaseen, Nahlah Abdulrahman Alkhalidi, and Hamed S Al-Raweshidy. She networks: Security, health, and emergency networks traffic priority management based on ml and sdn. *IEEE Access*, 10:92249–92258, 2022.

[116] Alessio Zappone, Marco Di Renzo, and Mérouane Debbah. Wireless networks design in the era of deep learning: Model-based, ai-based, or both? *IEEE Transactions on Communications*, 67(10):7331–7376, 2019.

[117] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. SIGCOMM '12, page 139–150, New York, NY, USA, 2012. Association for Computing Machinery.

[118] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Iisy: Hybrid in-network classification using programmable switches. *IEEE/ACM Transactions on Networking*, 32(3):2555–2570, 2024.

[119] Changgang Zheng and Noa Zilberman. Planter: seeding trees within switches. SIGCOMM '21, page 12–14, New York, NY, USA, 2021. Association for Computing Machinery.

[120] Chen Zhong, Ziyang Lu, Mustafa Cenk Gursoy, and Senem Velipasalar. A deep actor-critic reinforcement learning framework for dynamic multichannel access. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1125–1139, 2019.