Title Time Slot based Heavy Changer Prediction on P4 Switches

Student Name Chen Yiming

Student No. 2020103006

Major Computer Science & Technology

Supervisor Cui Lin

Date(dd/mm/yyyy) 04/05/2024

暨 南 大 学

本科生毕业论文

论文题目 P4 交换机上基于时隙的 Heavy Changer 预测方法

学　　院 国际学院

学　　系

专　　业 计算机科学与技术

姓　　名 陈奕铭

学　　号 2020103006

指导教师 崔林

2024 年 5 月 4 日

Time Slot based Heavy Changer Prediction on P4 Switches

Abstract: Heavy changer is flows change dramatically between two consecutive periods exceeds a specific threshold and can be viewed as a signal of congestion or malicious attacks. Identifying heavy flows (e.g., heavy hitters and heavy changers) in network traffic helps to improve network performance. It has many benefits to perform prediction of heavy changers within the data plane including eliminating extra communication between the control plane and the data plane and detecting the very beginning packages arrived at the switches. However, it is challenging due to extremely limited resource availability and the stringent requirements of fast packet processing. There are several proposed methods and systems in the research community for conducting network measurements on high-speed networks, yet there currently exists no advanced method to predict heavy changes in the data plane of programmable switches. In this thesis, we introduced pChanger, a programmable heavy changer prediction method based on time slots. This approach enables data planes on programmable switches to predict heavy changers by balancing performance and resource utilization, employing supervised machine learning (ML) techniques. In our evaluation, we first tested pChanger with traces and deployed it on bmv2 software P4 switches. The evaluation results indicate that pChanger achieved an accuracy of 99% and a recall of 75% on the data plane. We also demonstrate its near-linear runtime for predictions.

Key Words: Machine learning (ML), P4, heavy changer, programmable data plane, time slot

P4 交换机上基于时隙的 Heavy Changer 预测方法

摘 要：Heavy changer 是 heavy flow 的一种，指的是两个连续时间周期内流量剧烈变化且变化量超过特定阈值的流量。这类流量通常被视为网络阻塞或恶意攻击的信号，因此识别网络中的 heavy changer 有助于改善网络性能。然而，由于数据平面的计算资源极其有限，并且在交换机这种场景下需要对数据包进行快速处理，这项工作极具挑战性。学界提出了一些在可编程交换机上进行网络测量的方法和系统，但目前尚无方法在可编程数据平面预测 heavy changer 流。一些方法需要在控制平面和数据平面之间进行通信，从而引入了不可避免的延迟。一些方法需要在一些数据包之后才能判定到达交换机的新流是否为 heavy changer，容易造成不必要的网络拥堵。在本文中，我们提出了 pChanger（一种基于时隙的可编程 heavy changer 预测方法），它允许可编程交换机上的数据平面在性能和资源之间权衡的情况下，基于有监督的机器学习（ML）方法执行 heavy changer 预测。在评估过程中，我们首先在数据集上测试了 pChanger 的性能，随后在 bmv2 软件 P4 交换机上部署了 pChanger。结果表明，pChanger 的准确率达到了 99%，召回率达到了 75%。我们还展示了其近乎线性的预测运行时间。

关键词：机器学习（ML）；P4；重大变化流；可编程数据平面；时隙

Contents

1. Introduction

As network usage and traffic volume surge, congestion and anomalies are increasingly likely to occur due to sub-optimal network configurations and malicious activities. These issues can undermine the stability and security of networks, potentially resulting in significant economic losses.

The identification of heavy flows such as heavy hitters and heavy changers matters in optimizing network capacity allocation, detecting and mitigating network attacks, and efficiently scheduling traffic. A Heavy Changer (HC) is defined as a flow in which the variation in size between two successive epochs exceeds a specified threshold. Such variations can indicate potential congestion or malicious activities within the network (Zheng et al., 2022). By pinpointing these significant traffic flows, network administrators can better allocate resources, safeguard the network from potential threats, and ensure smoother and more reliable network operation. This proactive approach to network management not only boosts performance but also contributes to the overall stability and security of the network infrastructure.

The programmable data plane, supporting high packet forwarding speed and flexible programmability, is becoming a trend (Hauser et al., 2023). Programmable data planes, utilizing languages like P4 (Bosshart et al., 2014), offer a transformative approach to packet processing, enabling rapid protocol development and reducing resource use. P4-enabled switches provide unmatched speed, especially for small packets (He et al., 2018), and adaptability in dynamic network conditions, unlike fixed-function middleboxes. Unlike traditional SDN's centralized control, P4 switches handle packets autonomously, minimizing delays due to controller overload. This flexibility and efficiency make programmable data planes highly advantageous for modern network environments.

Implementing programmable hardware data planes involves challenges such as limited memory and computational resources. These constraints restrict the data plane's capacity to store detailed network states, adversely affecting state-dependent functions like five-tuple-based load balancing. Programmable hardware data planes encounter several challenges related to implementation, including constraints on memory and computational resources. These limitations can hinder the ability of the data plane to maintain

comprehensive network states, which is crucial for the performance of state-dependent network functions like load balancing that rely on five-tuples. Also, the scarcity of computational resources curtails the sophistication of network function designs and hampers the feasibility of the implementation of multiple functions at the same time. Although numerous methods have been developed to detect heavy changers, detection at the control plane level necessitates extra interaction between the data and control planes. This leads to inevitable heavy communication overhead and latency between these two planes. Some approaches have implemented heavy changer detection directly on the data plane using sketch-based techniques; concurrently, while most existing machine learning-based methods predominantly focus on predicting heavy hitters, very few have employed machine learning models to predict heavy changers.

Due to the variability of network traffic over time, we can dynamically predict heavy changers using time slots. Moreover, using time slots allows for more accurate confinement of heavy changers within a single time slot. In this paper, we introduce pChanger, a decision tree-based model designed for the rapid prediction of heavy changers in the data plane of P4 switches. The data shows that pChanger accurately detects heavy changers with a success rate of 91%. Furthermore, pChanger's sole operation within the data plane reduces superfluous exchanges and wait times between the control and data planes therefore eradicating unneeded communication burdens and latencies with the controller, thereby enhancing overall efficiency. This model facilitates efficient memory utilization while maintaining high throughput.

In summary, our contributions are as follows:

- We recap foundational technologies pertinent to this dissertation and delineate the current state of research, encompassing definitions of heavy changers, software-defined networking, programmable data planes, the P4 language, and various network traffic monitoring techniques, including methods for heavy flow detection, see Section 2.
- We detail the architectural design and prediction algorithms of pChanger and discuss the implementation of pChanger within the resource limitations of programmable hardware switches, see Section 3.
- We explored the performance and advantages of pChanger via trace-driven

experiments and simulator experiments with some sketch-based methods, see Section 4.

- We consolidate our conclusions, discuss the study's limitations, and propose future research avenues, as detailed in Section 5.

2. Backgrounds and relative works

2.1 Backgrounds

2.1.1 Heavy changer

Considering a stream of packets, dented by a key-value pair (x, v_x), where x is a key drawn from a domain $[n] = \{0, 1, \cdots, n-1\}$ and v_x is the value of x (Tang, Huang, Q. and Lee, 2019). In the network measurement case, x refers to a flow identifier, which is some kind of combination between source/destination information and protocol information (e.g., source/destination address pairs, 5-tuples), and v_x is either one (for packet counting) or the packet size (for byte counting). Conduct measurements at regular time intervals, which are called epochs or slots in some works (Tang et al., 2019). In this work, this kind of interval is called a time slot.

Heavy changers are defined as follows. Let ϕ be a pre-defined fractional threshold between 0 and 1, which is used to mark the heavy flows from network traffic. Let $S(x)$ be the sum (of all v_x's) of flows x in an epoch and S be the total sum of all flows in an epoch (i.e.,$S = \sum_{x\in[n]} S(x)$). Let $D(x)$ be the absolute change of $S(x)$ of flow x, while D is the total absolute sum of all flows in an epoch (i.e., $D = \sum_{x\in[n]} D(x)$) (Tang et al., 2019). A heavy changer is a flow whose difference in $v(x)$ (consider packet size as $v(x)$ in this work) between two consecutive epochs exceeds that threshold, which can be viewed as a signal of congestion or malicious attacks (Zheng et al., 2022).

2.1.2 Software-defined networking and programmable data planes

Conventional networking equipment, whether purely software-based or built on specialized hardware (like switches, edge routers, or security gateways), typically merges both the control and data planes within the same unit. These planes are crucial to the structure of any networking system. The data plane handles the actual transmission and routing of data packets, whereas the control plane oversees and directs the data layer's operations. Modifying network

settings necessitates adjustments in both layers.

On the other hand, Software-Defined Networking (SDN) revolutionizes network management and design by decoupling these layers, as illustrated in Figure 2-1, providing enhanced adaptability and versatility in arranging network configuration.

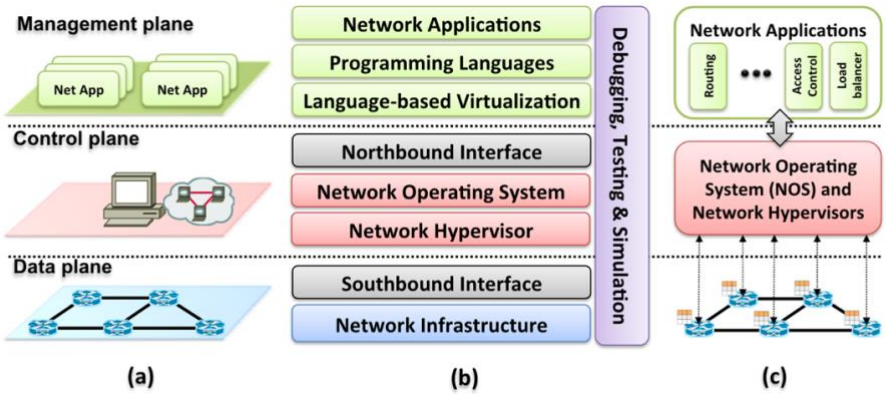

Figure 2-1 Software-defined network architecture depicted through planes, layers, and system configuration (Kreutz et al., 2015)

This decoupling allows for centralized control, wherein a network administrator can shape traffic and deploy services across the entire network from a single, logical point, enhancing efficiency and flexibility (Kreutz et al., 2015). SDN's architecture also facilitates network programmability, enabling automation and making it easier to integrate and manage resources in dynamic computing environments, such as cloud services and data centers.

In an SDN model, the control plane resides in the SDN controller, which communicates with the data plane devices (like switches and routers) via standardized protocols such as OpenFlow. This allows the controller to direct data plane devices on how to handle network traffic. SDN's centralized control provides a global view of the network, which can simplify and optimize resource allocation, reduce complexity, and support rapid innovation and evolution of network functionalities including advanced network management. This dynamic management enables more efficient resource use, leading to cost savings and reduced operational expenses. As a network paradigm, SDN is transformative, propelling networks into a new era of flexibility, control, and innovation.

The programmable data plane introduced by SDN empowers network engineers to create tailored rules for managing data packets across devices like switches and routers. It offers

flexibility that outpaces the static nature of traditional network infrastructures (Kreutz et al., 2015). Unlike conventional data planes with hardwired packet processing, the adaptable data plane can dynamically adjust to various network demands. This means, as outlined in the provided article, that network traffic is not just directed by static, predefined rules but can be intelligently managed based on a range of criteria, from traffic signatures to specific user requirements (Kreutz et al., 2015). The centralized SDN controller plays a pivotal role, conversing with the data plane via protocols like OpenFlow (McKeown et al., 2008). This centralized command center allows for seamless, holistic network adjustments, aligning with the overarching goals of efficiency and agility highlighted in the text (McKeown et al., 2008).

2.1.3 PISA and P4 language

OpenFlow, while instrumental in advancing SDN, does encounter certain constraints. It offers only Exact Match and Prefix Match for packet inspection, which may fall short in complex scenarios requiring more nuanced packet handling. Additionally, when dealing with multiple network domains, OpenFlow necessitates manual setups for each, which can be a drawback for networks needing smarter, more automated operations. To address these limitations, the Protocol-Independent Switch Architecture (PISA) has been introduced. PISA complements OpenFlow by providing a flexible and adaptable framework for implementing network protocols, thereby facilitating multi-domain management. The ability of programmable switches based on the Protocol Independent Switch Architecture (PISA) to execute data plane programs at line rate has opened up new opportunities for researchers and practitioners, spurring unprecedented innovation in network protocols and architectures (Gebara et al., 2020).

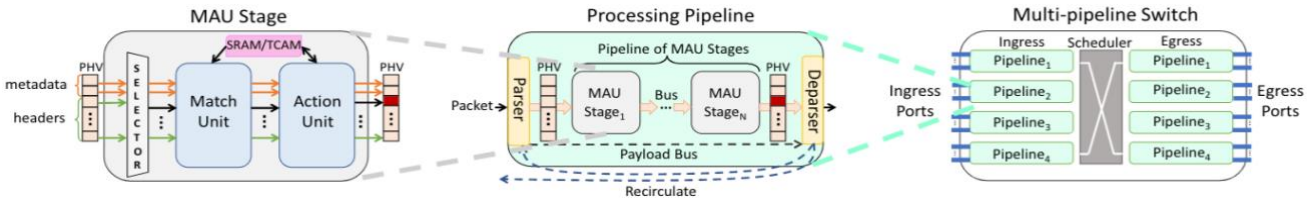

Figure 2-2 PISA-based switch, adapted from RMT architecture in different levels of detail: Match-Action Unit (MAU) stage within a single switch pipeline (left), switch pipeline of MAU stages (middle), and a switch with multiple pipelines (right) (Gebara et al., 2020)

The programmable data plane introduced by PISA has revolutionized network

management, allowing for the handling of a varied spectrum of traffic types and bolstering a multitude of protocols and packet configurations. This flexibility empowers network administrators to devise bespoke data processing protocols, facilitating an evolution toward a network ecosystem that is both highly automated and flexible. Concurrently, such advancements have forged the foundation for in-network computing, which gives rise to a spectrum of innovative applications—from caching and database queries to machine learning and consensus algorithms.

P4, an acronym for Programming Protocol-Independent Packet Processors, represents a sophisticated high-level language designed to program packet processors that operate independently of specific protocols (Bosshart et al., 2014). It grants network operators the ability to define packet processing functions via a dynamic, programmable syntax, affording the creation of tailored network functionalities and guidelines. P4's development has been notably advanced by integrating with PISA, utilizing the PISA framework to enhance its packet processing language's robustness and adaptability.

As depicted in Figure 2-3, the P4 abstract forwarding model includes several distinct stages, including a programmable parser and multiple stages of match+action. There are two examples of stages of match+action in the figure named ingress pipeline and egress pipeline. Initially, the parser converts the influx of packet streams into structured protocol headers along with corresponding metadata. The parsed packets then proceed to the ingress pipeline, where they are processed according to the specific matching criteria and actions predefined in the control program's rules. These rules dictate modifications and routing decisions for each packet within the switch, preparing them for further operation. Following this, the egress pipeline engages in further packet manipulations essential for the packet's transmission readiness. Culminating the process, the deparser reassembles the processed packet data into a bit stream suitable for outbound transmission.

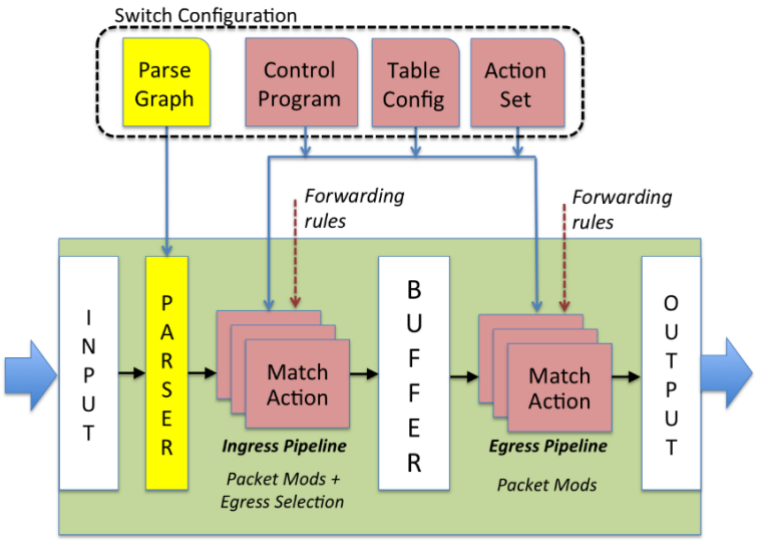

Figure 2-3 P4 Abstract Forwarding Model (Bosshart et al., 2014)

Moreover, P4 employs a target-agnostic abstraction known as the V1Model, which abstracts away the particulars of the underlying hardware—whether it's a switch or a router or built on NPU or FPGA technology. Thus, network developers can employ the specialized P4 language to program across a diverse array of hardware platforms. A P4 program is composed of several fundamental components: the parser, which decodes and organizes data from packet headers; the control flow, which establishes the rules for packet processing; the tables, which link actions such as rerouting or discarding packets to their respective header fields; and the actions, which implement the specified modifications to the packets. These elements work in concert to define and execute the processing logic required for network data handling within a P4-enabled device.

2.2 Relative works

Although detection and prediction encompass the scrutiny of network traffic, they utilize divergent techniques and aim for separate outcomes. The heavy changer detection approach leverages traffic characteristics to pinpoint data streams that dramatically change, followed by refinement through methods like classification and clustering. The objective of forecasting is to anticipate upcoming network traffic scenarios, enabling network administrators to implement necessary modifications to uphold network reliability and efficiency. Forecasting techniques typically deduce patterns and fluctuations in future network traffic by employing time series analysis, which normally includes examining and interpreting past traffic data,

thereby offering guidance for network resource distribution and strategic planning.

Some research works have performed heavy flow identification in different layers (see Table 2-1), while most of the works are based on data layers with sketch-based methods. For methods that perform heavy changer identification singly, statistics-based methods exist, separated into sketch-based methods and non-sketch-based methods. The heavy changer detection approach leverages traffic characteristics to pinpoint data streams' drastic change, followed by refinement through methods like classification and clustering.

Table 2-1 Summary of work related to heavy changer detection

Measurement methods/systems	**Location**	**Identification Method**
Count-Min Sketch (Cormode and Muthukrishnan, 2005)	Data Plane	Detection
Chain-Sketch (Huang, J. et al., 2023)	Data Plane	Detection
MV-Sketch (Tang et al., 2019)	Data Plane	Detection
LD-Sketch (Huang, Q. and Lee, 2014)	Data Plane	Detection
Tight-Sketch (Li and Patras, 2023)	Data Plane	Detection
OpenSketch (Yu, Jose and Miao, 2013)	Data Plane and Control Plane	Detection
Elastic Sketch (Yang et al., 2018)	Control Plane	Detection
Sketchlearn (Huang, Q., Lee and Bao, 2018)	Control Plane	Detection
pChanger (Our method)	Data Plane	Prediction

Count-Min Sketch (CM Sketch) (Cormode et al., 2005) is the very beginning of sketch implementation, which utilizes a 2D array and multiple hash functions. Specifically, it is characterized by a matrix of counters (width *w* and depth *d*), combined with *d* hash functions. Each hash function maps input elements to a column within a specific row of the matrix. For each element in the data stream, its corresponding counter in each row is incremented based on its hash values. The parameters d and w are chosen based on the desired error tolerance ϵ and failure probability δ, such that the sketch can guarantee that for any item, the estimated frequency is at least its actual frequency but does not exceed it by more than $\epsilon \times N$ (where N is the total count of items) with probability at least $1 - \delta$.

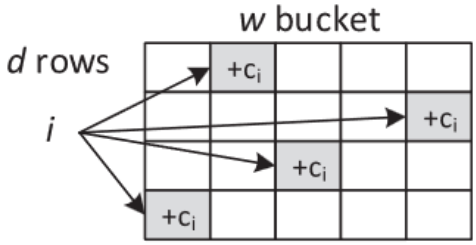

Figure 2-4 Overview of CM Sketch (Zheng et al., 2022)

While widely used for data stream summarization, CM Sketch faces several limitations including its inability to record flow keys, necessitating external input for flow identification. This approach compromises its generality and can lead to significant accuracy loss due to hash collisions in environments with dense network traffic. Moreover, its counters have a fixed size, which may lead to either overflow or underutilization depending on the flow's volume, and the requirement for multiple hash computations per packet introduces considerable computational overhead, particularly in software-based applications. So a lot of work has appeared to improve the sketch class methods in addressing these limitations. Tight-Sketch (Li et al., 2023) offers a refined approach for managing heavy item-oriented data stream mining within constrained memory environments. It introduces an innovative eviction strategy using probabilistic decay, specifically designed to handle high-speed data streams by favoring accuracy and memory efficiency. LD-Sketch (Huang, Q. et al., 2014) is an arrayed sketch structure that employs counter-based techniques, specifically designed to overcome the limitation of the Count-Min Sketch in recording flow keys. It aims to accurately detect both heavy hitters and heavy changers by leveraging a distributed architecture, ensuring efficient tracking and identification across multiple data streams. MV-Sketch (Tang et al., 2019), a compact and invertible sketch tailored for the efficient detection of heavy hitters and changers in network data streams, is structured as a two-dimensional array with r rows of w buckets, shown in Figure 2-5. Each bucket, identified by $B(i,j)$, holds a trio of fields: $V_{i,j}$ for the aggregate value sum, $K_{i,j}$ for the key of a potential heavy flow, and $C_{i,j}$ as an indicator count. Integrating a majority vote algorithm ensures minimal memory overhead and swift processing by leveraging static memory allocation, thus bolstering accuracy and throughput in identifying substantial traffic changes. ChainSketch's data structure is designed to optimize

heavy flow detection, effectively addressing memory efficiency and overestimation issues prevalent in network monitoring. The framework consists of r rows and w buckets arrayed to form a grid, where each bucket $B(i,j)$ houses two elements: the flow key $B(i,j).k$, indicating the candidate heavy flow's identifier, and $B(i,j).c$, a counter for the flow's size. This configuration is paired with a selective replacement strategy, reducing overestimation by only replacing flows when necessary, and utilizing hash chains to link flows across the structure. These features collectively enhance ChainSketch's ability to accurately and efficiently track heavy flows in network traffic, as depicted in Figure 2-6.

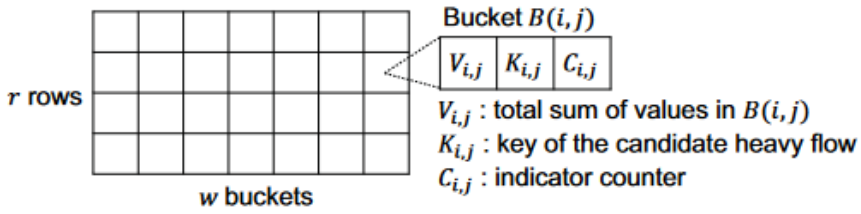

Figure 2-5 Data structure of MV-Sketch (Tang et al., 2019)

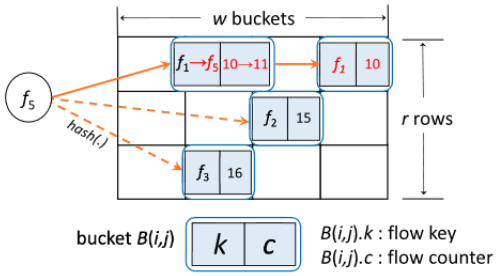

Figure 2-6 Basic idea of ChainSketch (Huang, J. et al., 2023)

Some articles also focus on detecting heavy flows in the control plane, or on working simultaneously in both the data plane and the control plane. OpenSketch (Yu et al., 2013) presents a flexible, software-defined network measurement architecture that features a streamlined data plane for deploying various sketch algorithms and a smart controller for configuration and analysis. Its three-stage data plane—hashing, classification, and counting—enables efficient traffic statistics accumulation without recording flow keys, while its controller simplifies measurement tasks through automatic sketch management and resource allocation. Elastic Sketch (Yang et al., 2018) introduces sketch compression

algorithms and merging strategies, which stands out as a two-part dynamic structure that not only adjusts to traffic conditions for precise network-wide measurements but also innovates with compression techniques that conserve bandwidth and enhance adaptability. SketchLearn (Huang, Q. et al., 2018) aims to simplify network management by using a multi-level sketch built with small sketches to analyze and isolate large data flows, ensuring smoother handling of smaller flows. This system adapts its settings dynamically, minimizing the dependency on specific configurations for accurate results. It features a dual architecture with a distributed data plane that processes packets at multiple nodes and a centralized control plane that breaks down data into three key components: 1) the large flow list, which identifies and logs large data flows, including their estimated frequency and associated errors; 2) the residual multi-level sketch, which maintains traffic statistics for the remaining small flows; and 3) the bit-level counter distributions, each distribution models the counter value of each level sketch in the residual multi-level sketch (Zheng et al., 2022). Suppose a multi-level sketch has $l+1$ levels when the bit number of the flow key is l, each level consists of a sketch featuring r rows and c column counters, all utilizing the same r hash functions. In this setup, the h_i hash function maps a flow key to the j^{th} column in the i^{th} row. Let $p[k]$ represents the probability of the k^{th} bit equals to one and $R_{i,j}[k] = \frac{C_{i,j}[k]}{C_{i,j}[0]}$ (Huang, Q. et al., 2018). The key property of SketchLearn's multi-level sketch is that in the absence of large flows, the $R_{i,j}[k]$ values, representing adjusted counter differences, follow a Gaussian distribution with a mean of $p[k]$. Utilizing this property, SketchLearn identifies and isolates large flows by comparing $R_{i,j}[k]$ to $p[k]$ until the data conforms to the Gaussian model. This process effectively segregates large from small flows.

Some non-sketch approaches propose methods for estimating flow sizes, which are either based on traditional mathematical and statistical methods or counter-based methods, such as BeauCoup (Chen et al., 2020) and LOGLOG (Durand and Flajolet, 2003). Although these methods do not directly implement detection for the specific application of heavy changers, they are equally applicable as part of the heavy changer detection process. In recent years, there has been work applying machine learning to the management of network traffic, particularly in making predictions about specific network flows, including tasks within the control plane and data plane (Xiong and Zilberman, 2019). Some efforts have begun

deploying machine learning predictors on controllers in SDN networks, using information from the first few packets of each flow to make predictions about specific network flows. Pouper et al. (2016) present a novel application of data mining in the control plane for predicting the size of network flows and detecting large flows (elephant flows), which carries out flow size inference employing three machine learning methods, online Bayesian matrix matching, neural networks, and Gaussian process regression, through online machine learning techniques, which are crucial for improving network functions like routing, load balancing, and scheduling. Deepa, Sudar and Deepalakshmi (2018) introduced a hybrid machine-learning model for detecting DDoS attacks on controllers in an SDN environment. By combining Support Vector Machine (SVM) and Self-Organizing Map (SOM), this model can predict the size of flows at their onset without any modification to applications or end hosts, effectively improving network scheduling and routing.

Nonetheless, implementing a machine learning-based model within the control plane leads to inevitable increases in communication overhead and time lags between the data plane and the control plane. To address this issue, some works have emerged that apply simplified machine learning models directly within the data plane, which helps to mitigate the interaction delays and overhead between the data plane and the control plane. Xavier et al. (2021) present a method for deploying machine learning models into programmable switches for real-time, in-network traffic classification, which leverages P4 language capabilities to implement decision trees that operate directly on the data plane. Busse-Grawitz et al. (2022) introduced pForest, a system designed for ASAP in-network classification leveraging machine learning models on programmable data planes. The system dynamically adapts its feature selection and classification strategy throughout a flow's lifecycle, employing a sequence of tailored random forest models and switching among them in real time on a per-packet basis. Carvalho et al. (2021) introduced DataPlane-ML, a system that employs machine learning techniques directly on the data plane of SDNs to detect DDoS attacks. This system utilizes P4 programmable switches and machine learning libraries to analyze traffic and identify potential threats without overloading the SDN controller.

The existing machine-learning approaches have not focused on the task of detecting heavy changers. Our model is able to directly predict heavy changers on the data plane,

thereby making the data plane prediction capable for heavy changers. Compared to other data plane-based methods for detecting heavy changers, our model achieves higher accuracy and recall under the same memory constraints and it identifies faster than detection methods.

3. pChanger design

3.1 Motivation and challenges

Implementing heavy changer prediction within the programmable data plane eliminates extra communication between the control plane and the data plane and enables real-time detection of initial packets arriving at switches. However, there are two major challenges in designing the pChanger model. The very first challenge relates to the skewed distribution of data during the model's training phase. It's crucial to account for how this imbalance might skew the model's ability to correctly classify data. The second issue can be seen as coupled to the limitations of the data plane when utilizing a P4 switch. Considering the constrained memory and processing power of programmable hardware data planes, any developed model must be streamlined to ensure it can operate effectively within these confines.

a) Issue of Skewed Data Distribution: In real-world networks, an extremely small fraction of data flows are responsible for unexpectedly high changes in traffic volume in a short duration. While the precise proportions are related to the threshold we define, this stark disproportion often leads to issues of data imbalance in machine learning models tasked with analyzing network traffic. The result is an overrepresentation of normal data flows in the sample set, which can skew the training outcomes of the models, making them less effective at identifying significant traffic changes.

b) Constraints of Data Plane Resources: The programmable switch data planes based on P4 face several limitations, including restricted memory, limited processing capabilities, and constrained bandwidth. Memory constraints are particularly prevalent owing to the requirement for the data plane to hold comprehensive forwarding tables, storage caches, and other organizational data, making memory a critical resource. These constraints often manifest as a capped memory capacity, leading to potential performance dips and data packet loss when exceeded. In the context of predicting large data flows, the memory capacity restricts not only the number of network flows that can be monitored but also the size of the

model. The constraints on processing power, influenced by the speed and number of processors within the switch, often lead to the delay and loss of packets. Limitations on bandwidth, resulting from restrictions on the switch's capacity, can intensify problems related to packet loss and delays. Moreover, because the data plane is unable to perform division and floating-point operations significantly restricts the deployment of many machine learning techniques that rely on floating-point computations. It necessitates the exploration of alternative methods.

3.2 System overview

The pChanger methodology (Programmable Heavy Changer Prediction based on Time Slot) is designed to precisely forecast the transition moments of data flows from mouse flows to elephant flow or from elephant flow to mouse flow within a designated time slot range. This approach is structured around two primary phases. Initially, Python's dpkt library is utilized to parse pcap files for feature extraction. This stage often encounters a significant challenge of data imbalance among the extracted features, necessitating the employment of data cleaning and sampling techniques. To tackle the challenge of imbalance, methods such as OneSidedSelection and RandomUnderSampler from the imbalanced-learn (imblearn) library are utilized to alleviate the imbalance concerns effectively. Subsequently, the DecisionTreeClassifier from the scikit-learn library is employed to both train and select an appropriate prediction model based on the cleansed data. The second phase involves translating the refined model into p4 code, which is then deployed onto the data plane of a P4-enabled switch. This deployment facilitates the on-the-fly prediction of heavy changer flows within the network infrastructure, enabling a more dynamic and efficient network traffic management by pinpointing potential bottlenecks or high-demand periods in advance.

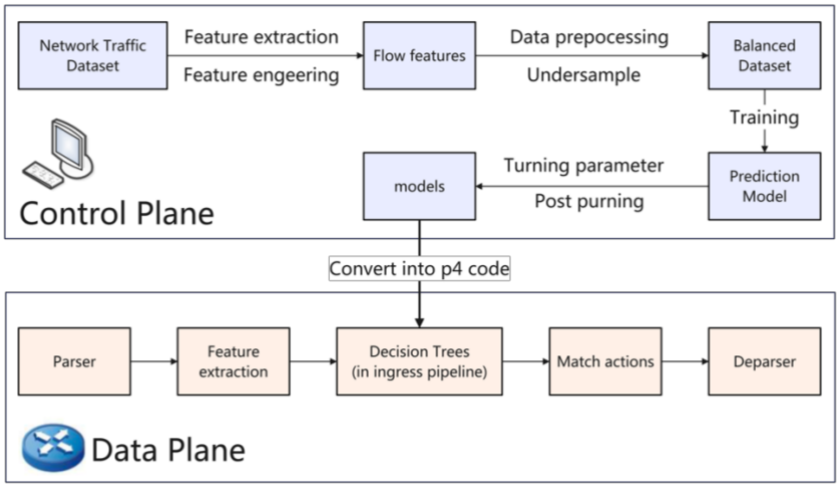

Figure 3-1 pChanger Procedure

pChanger outlines the heavy changer issue through various parameters: *slot_size* represents the duration of time slots, *ahead* indicates the count of future slots, and the heavy changer threshold ϕ, constrained by $0 < \phi < 1$. For heavy changer threshold definition, we follow the works MV-Sketch (Tang et al., 2019) and Chain-Sketch (Huang, J. et al., 2023), define flows whose differences between two successive time slots exceed the threshold ϕD as heavy changer (see Section 2.1.1).

3.3 Feature extraction

3.3.1 Features

Previous studies in network traffic analysis have gained a variety of valuable features, which can be divided into two parts, the packet-level features (e.g., IP address, payload length) and the flow statistics (e.g., packet count, duration) (Nguyen and Armitage, 2008). Our work takes both types of features into model building. We have removed potentially unique identifiers to prevent excessive branching in the decision tree, such as names, destination IPs, and source IPs. Additionally, we have eliminated timestamp-related information, as determining whether it is a heavy flow should not be closely tied to the time when the flow occurs. It is particularly important to note the feature 'proto'. Its values adhere to the protocol numbers maintained by the Internet Assigned Numbers Authority (e.g. 6 for TCP) (IANA, 2024). Table 3-1 displays the remaining features.

Table 3-1 Network Traffic Features selected by pChanger

Feature Name	**Description**
Dport	Destination Port
Sport	Source Port
Proto	Protocol type of the current IP packet
Length_max	Maximum length of packets for the given flow
Length_min	Minimum length of packets for the given flow
Length_mean	Average length of packets for the given flow
Total_length	Cumulative packet length of the respective flow
IAT_max	Maximum inter-arrival time of packets for the respective flow
IAT_min	Minimum inter-arrival time of packets for the respective flow
IAT_mean	Average inter-arrival time of packets for the respective flow
ACK_flag	Counter for ACK flags of the respective flow
FIN_flag	Counter for FIN flags of the respective flow
SYN_flag	Counter for SYN flags of the respective flow
PSH_flag	Counter for PSH flags of the respective flow
RST_flag	Counter for RST flags of the respective flow
ECE_flag	Counter for ECE flags of the respective flow
Packet_count	Counter for packets associated with the respective flow

3.3.2 Feature extraction procedure in dataset

pChanger, leveraging the dpkt module in Python, efficiently parses and constructs packets from network traffic in *pcap* format, while also outlining essential TCP/IP protocols. This tool captures network flow features, which it assigns to specific slots in a data frame. Notably, it maintains these features in a non-cumulative manner to ensure that the timing of each flow is accurately synchronized.

After capturing the characteristics of all flows within two given consecutive time slots, pChanger identifies flows whose flow change exceeds the total flow change above the specified threshold ϕ and labels them as heavy changers. Subsequently, for flows that will change hugely, pChanger gathers their features from prior epochs as positive examples, based

on the *ahead* parameter. These are then categorized as flows likely to vary. Conversely, for other flows occurring in the current slot, pChanger gathers the features from the first previous slots as negative samples. pChanger immediately discards flows that lack features in previous slots, signifying their non-existence during those periods.

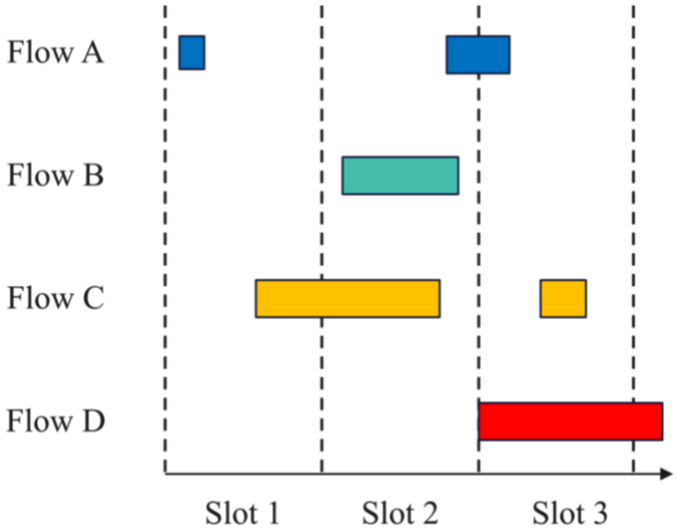

Figure 3-2 Sample diagram demonstrating the process of feature extraction

Take Figure 3-2 as an example and let us concisely explain the feature extraction process. Setting *ahead*=1 and slot 2 as the current time slot, with only four types of flows in the network and current time. The pChanger captures features from slot2 and determines that the change in FlowB and FlowC from slot1 to slot2 exceeds the threshold ϕ compared to the changes in all four flows during the same interval. Since FlowB did not have its features prematurely captured from the previous interval, solely the characteristics of slot1's FlowC were identified as positive samples, marking it as an impending change flow. For other flows such as FlowA that appear in the current interval, pChanger captures their features from slot1 as negative samples.

Assuming *ahead*=2, when pChanger captures features from slot3, it identifies FlowB, FlowC, and FlowD as heavy changers. Since FlowD did not appear in the previous two intervals, only slot 3's FlowD is marked as a significant changer. For FlowB, slot1's FlowB is marked as an impending change flow, and for FlowC, both slot1 and slot2 are marked as impending change flows.

3.4 Data preprocessing

To get a better model we have tried 96 sets of parameters for the feature extraction part on the real-world dataset UNIV1, the more detailed information is shown in Section 4.1. We used *ahead* = 1, 2, 3, and 4 with *slot_size* = 2, 3, 4, and 5 to predict heavy changers at 6 different thresholds, i.e. ϕ= 0.0008, 0.005, 0.02, 0.05, 0.1, and 0.2 respectively.

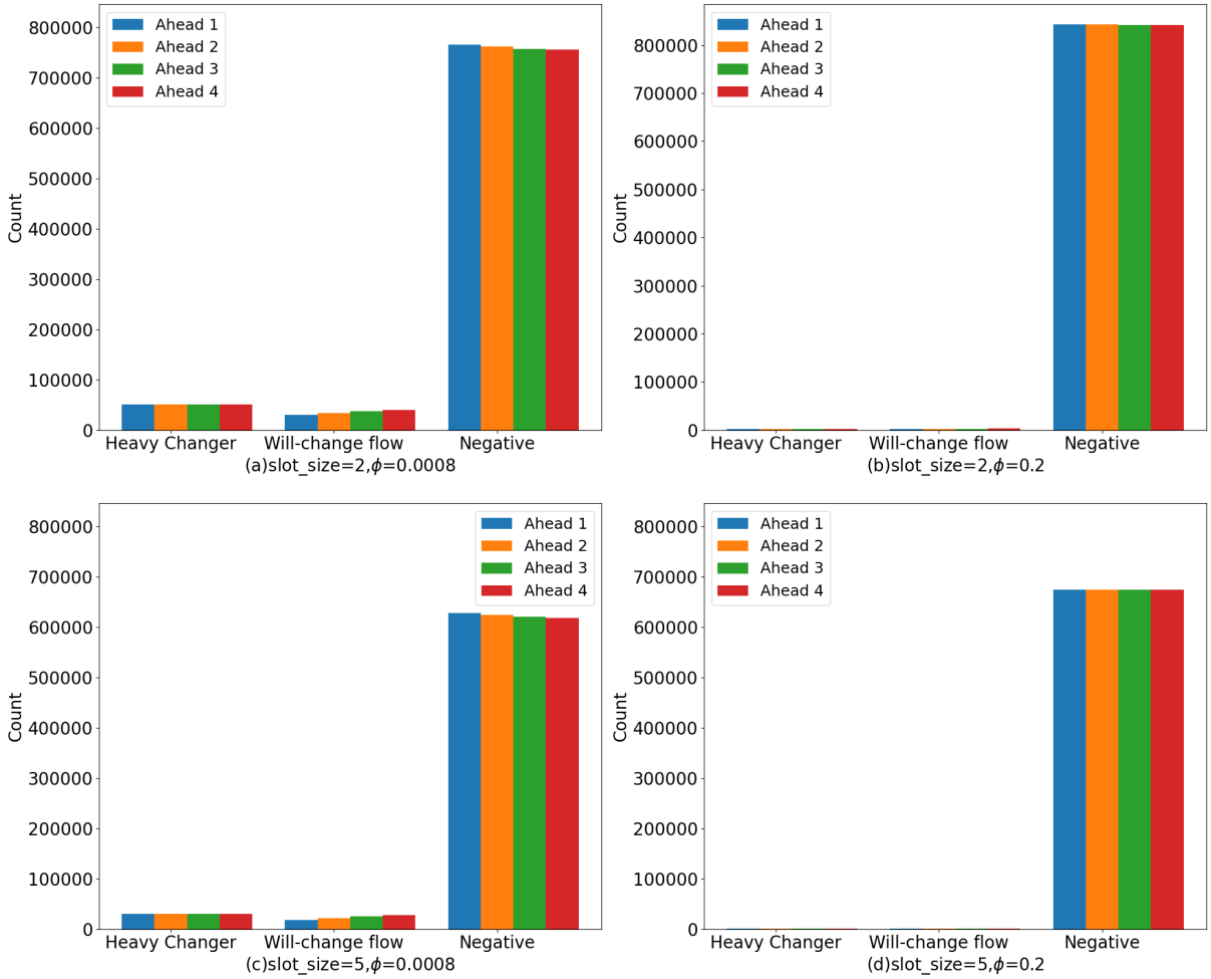

Figure 3-3 Distribution of samples prior to data preprocessing

3.4.1 Imbalanced data problem

Collect samples of heavy changer and will-change flow as the positive samples and we count the reverse imbalanced ratio as $Reverse\ Imbalance\ Ratio = \frac{Number\ of\ samples\ in\ minority\ class}{Number\ of\ samples\ in\ majority\ class}$. We can find that the maximum ratio is 0.2023 and appears when slot_size = 2, ϕ = 0.0008, and ahead = 4 with the minimum ratio 0.0093 appearing when slot_size = 2, ϕ = 0.05, and ahead = 1. From that, we can find the dataset exhibits a significant issue of data imbalance, which neither sampling nor cost-sensitive learning

approaches can adequately address. This is due to the complexity of the decision tree model generated from the highly imbalanced data, which surpasses the memory constraints of the p4 switch, rendering it infeasible to compile and deploy the model on the data plane for prediction purposes. In response to this challenge, pChanger applied two strategies: one eliminating flows with fewer than 10 packets and the other eliminating flows with fewer than 20 packets. Then, while training the decision tree with the same parameters, each strategy was applied separately. The primary justification for the elimination of these data features is that such flows only begin transmitting to the switch towards the end of a time slot, resulting in a minimal number of packets being recorded as flow features for that particular slot. The sparse data from these flows contributes to high information entropy, substantially increasing the risk of misclassification due to the insufficient information they provide. The opaque parts of Figure 3-4 show the percentage of remaining traffic in the original traffic after eliminating flows with fewer than 20 packets. A significant portion of ordinary traffic was removed.

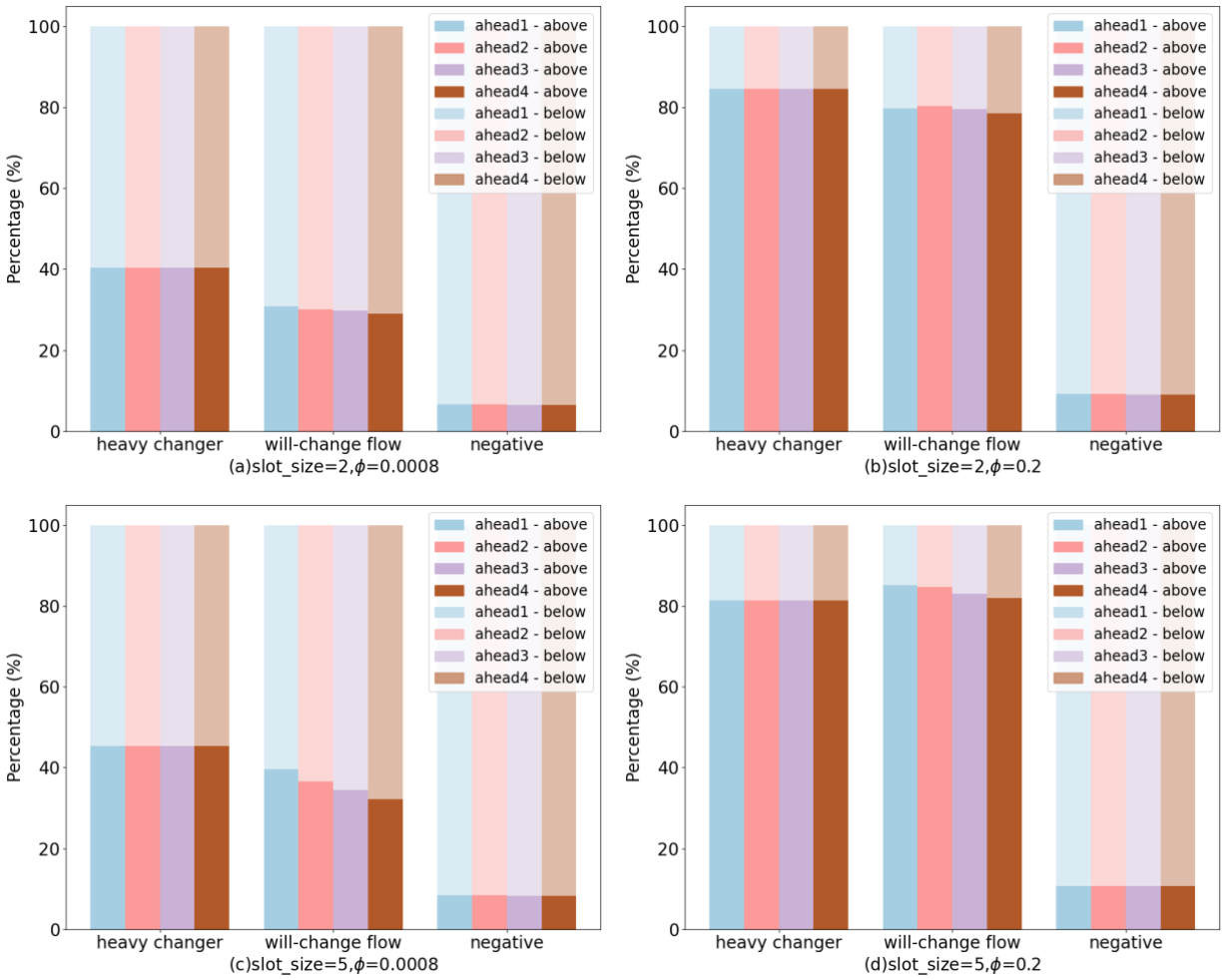

Figure 3-4 The Proportion of Samples with less than 20 packages

Figure 3-5 depicts the ratio of false positives in the dataset after cleansing. Through observation, it is possible to identify the highest, lowest, and mean values. Eliminating traffic features that include fewer than 20 packets significantly reduces the number of negative samples; nonetheless, a considerable imbalance remains within the dataset. Although the decision trees trained with such data demonstrate high accuracy, they are overly complex. Attempts to deploy overly large decision trees on the data plane have proven unsuccessful. To tackle this issue, pChanger utilizes the One-Sided Selection (OSS) method and random under-sampling, effectively decreasing the volume of data and significantly alleviating the dataset imbalance. Additionally, this approach manages to reduce the complexity of the tree without substantially compromising accuracy.

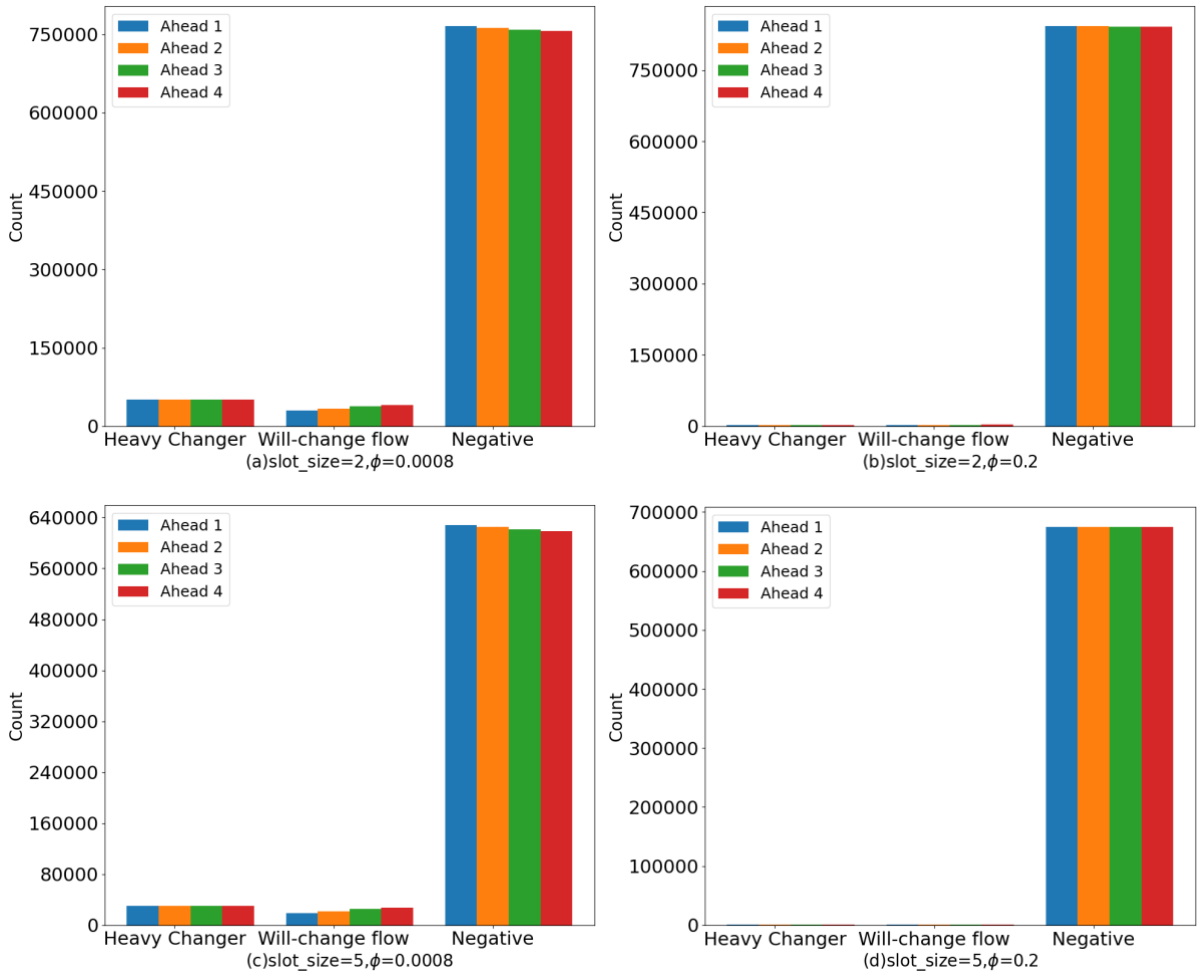

Figure 3-5 Sample Distribution after Data Processing

3.4.2 the One-Side Selection technique

Figure 3-6 presents some positive examples, multiple negative examples, and conjectural

hyperplanes that differentiate between the positive and negative samples. In this figure, the negative samples are categorized into four distinct groups: 1) Samples affected by class-label noise, such as the point located in the bottom-left corner; 2) Borderline examples situated near the boundary separating the positive and negative regions, which are prone to misclassification with slight attribute noise; 3) Redundant samples that can be represented by other examples, exemplified by those in the upper-right corner; 4) Safe examples that are valuable and should be retained for future classification tasks.

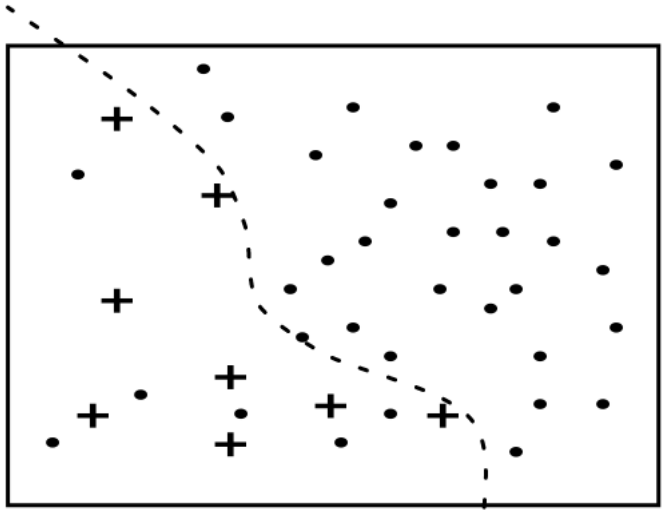

Figure 3-6 Unbalanced Sample Distribution example (Kubat and Matwin, 1997)

Redundant examples do not necessarily impair classification accuracy; however, they do contribute to increased classification expenses. The objective of One-Sided Selection (OSS) is to remove as many redundant examples as possible. For detecting borderline examples and those affected by class-label noise, the concept of Tomek links (Tomek, 1976) proves effective. The concept behind Tomek linking is based on selecting two samples, x, and y, with distinct labels. In cases where no sample z satisfies $\delta(x,z) < \delta(x,y)$ or $\delta(y,z) < \delta(y,x)$, the pair (x,y) qualifies as a Tomek link. Samples that form part of Tomek links are thus classified as either marginal or noisy.

Efforts to minimize the number of superfluous examples can be seen as the creation of a consistent subset C from the training set S. By definition, if a subset C, which belongs to S, classifies the examples within S accurately under the application of the 1-NN rule, it is considered aligned with S. In this specific context, our objective is not to produce the smallest

possible C but to ensure that the collection of negative examples remains manageably small. To achieve this, we employed a modified version of Hart's technique (Hart, 1968). Next, the samples in C underwent the 1-NN rule to reassess the entire collection S. Any training samples that were incorrectly classified during this process were subsequently incorporated into C. The One-sided Selection process is outlined as follows: Initially, this method starts by creating a subset C that matches the training set, thereby reducing the count of redundant negative samples. It then continues to eliminate negative samples engaged in Tomek links. This strategy effectively refines the training set into a cleaner subset T by eliminating noisy and marginal samples.

Algorithm One-sided Selection

Input:

S: the initial training set.

C: a stable subset composed of every positive from S alongside a negative chosen randomly

Output:

T: reduced cleaner training subset

1. **For** sample s in S **do**
2. Classify s with *C* using 1-NN and verify label accuracy
3. **If** s is misclassified, move s into *C*
4. Eliminate all negative examples in C involved in Tomek links
5. $T \leftarrow C$

In the use of the OneSidedSelection technique from the imblearn library for undersampling, it was observed that the effectiveness of this method on lightweight flows is limited. Despite the adjustment of various parameters, the undersampling results for lightweight flows did not meet the anticipated outcomes. This suggests that most redundant samples in the lightweight flow datasets had already been removed during earlier data cleansing processes. Consequently, the remaining samples predominantly consist of non-redundant, or 'safe', examples. This outcome underscores the significance of the initial data cleansing phase in the preprocessing of datasets for analysis.

3.4.3 Undersampling method

Considering the issue of imbalanced distribution in dataset S, let's assume there are two classes. The set of minority samples is referred to as the minority class $S_{min} \subset S$, while the set of majority samples is defined as the majority class $S_{maj} \subset S$. In this case, we have $|S_{maj}| \gg |S_{min}|$ and $|S| = |S_{maj}| + |S_{min}|$. The purpose of sampling methods is to balance the number of minority and majority classes and generate dataset $|S_{new}|$. It generally falls into two categories: oversampling and undersampling.

In oversampling, we define the new dataset after sampling as $|S_{new}|$, which is balanced by adding samples $|S_{over}|$ to the minority class, such that $|S_{\text{new}}| = |S_{\text{min}}| + |S_{\text{maj}}| + |S_{\text{over}}|$. On the other hand, undersampling achieves balance by removing majority class samples, resulting in $|S_{\text{new}}| = |S_{\text{min}}| + |S_{\text{maj}}| - |S_{\text{under}}|$. Due to the complexity of machine learning models constructed with oversampled data, which can be challenging to operate in resource-constrained data plane environments, we opt for undersampling methods here. We use random-based methods (The imbalanced-learn developers, 2024) to remove samples of the majority class.

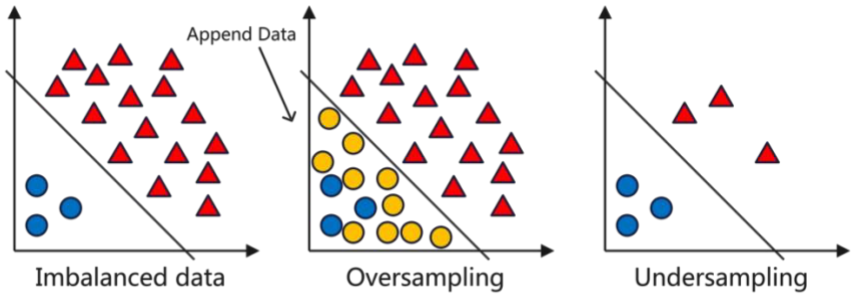

Figure 3-7 Result of different sampling methods

Figure 3-8 shows an example of a decision tree trained using One-Sided Selection and RandomUnderSampler. The resulting tree has a small number of nodes and depth, demonstrating that with these methods, we can obtain a decision tree that can be deployed on a programmable data plane.

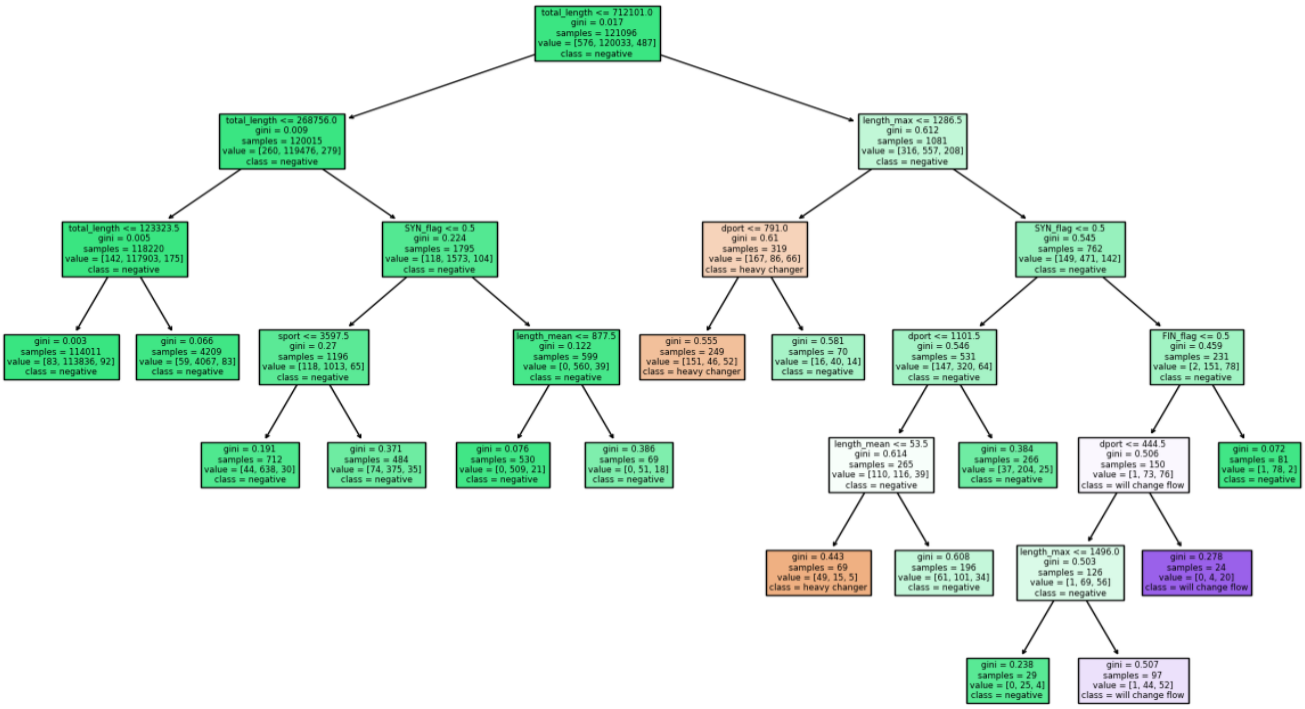

Figure 3-8 Decision Tree trained with One-Sided Selection and RandomUnderSampler

3.5 Prediction model design

3.5.1 Prediction model selection

pChanger opted for decision trees as its predictive model. The decision tree was chosen based on its straightforward structure, characterized by simple components such as conditional statements, and uncomplicated operations that do not require floating-point calculations. This simplicity facilitates easier implementation of the model (Xiong et al., 2019). Unfortunately, decision trees are inherently vulnerable to unbalanced data distributions. Decision trees struggle with unbalanced data, leading to overfitting and poor representation of smaller classes, as they tend to favor larger classes in their branching, akin to what occurs with the 1-Nearest Neighbor (1-NN) classifier.

Pruning, which involves removing leaves with class probability estimates falling below a certain threshold, can enhance the generalization of decision trees. However, in contexts where data is unbalanced, this pruning process tends to disproportionately eliminate leaves that represent less frequent, albeit potentially critical, concepts. This can further exacerbate the challenges of modeling with an unbalanced dataset. Research has demonstrated that while pruning decision trees in response to unbalanced data may seem beneficial, it does not improve performance, and using unpruned trees under such conditions is equally ineffective

(Huang, N.-F. et al., 2013). Whereas, pruning fails to solve the core problem. After nodes are pruned, they often still contain samples from both categories. The practice approach usually is to label each leaf based on the dominant class present in that area. In situations where positive samples are sparse, areas with a combination of positive and negative samples usually end up labeled as negative. Unless modifications are made to the algorithm, branches are not recognized for their potential influence from positive examples. This approach overlooks a fundamental issue: negative examples are often outnumbered by positive ones, leading to most regions being labeled negatively. To tackle this persistent issue of data imbalance, pChanger opts for an alternative strategy by implementing cost-sensitive decision trees. These trees are designed to weigh the classes differently, which helps in balancing the influence of the less frequent positive examples in the decision-making process.

3.5.2 Model implementation

pChanger employs DecisionTreeClassifier from the scikit-learn module. The Classification And Regression Tree algorithm, commonly abbreviated as CART, one of the optimized versions of it is implemented by scikit-learn (Pedregosa et al., 2011). Unlike traditional methods that rely on information entropy to determine the most effective feature splits, CART adopts the Gini coefficient, also referred to as Gini impurity. Both metrics effectively quantify the amount of information or impurity within a dataset. However, the Gini coefficient is particularly favored in this application because it significantly reduces computational demands. This reduction is primarily due to the fact that calculating the Gini coefficient does not require logarithmic operations, thereby making it less computationally intensive.

For each candidate split feature k and corresponding threshold t_k, the CART algorithm formulates a strategy that seeks to reduce the Gini impurity on each side of the division, weighted by the number of instances in each resulting node (Breiman et al., 2017). The objective function for classification is designed to achieve the lowest weighted Gini impurity, thereby optimizing the homogeneity of the classes post-split, which is:

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

$$\text{where} \begin{cases} G_{left/right} & measure\ the\ impurity\ of\ the\ left/right\ subset \\ m_{left/right} & is\ the\ nbumber\ of\ instances\ in\ the\ left/right\ subset \end{cases}$$

It is important to recognize that each split made by the CART algorithm follows a greedy

approach, and thus, it does not guarantee a globally optimal solution. This limitation arises because determining the optimal decision tree configuration is an NP-complete problem, which involves complexities beyond polynomial time computations. Therefore, the goal is to identify a feasible (though not necessarily optimal) solution. The process of tree splitting in CART is halted either when the tree reaches a predetermined maximum depth, determined by the *max_depth* parameter setting, or when further splits do not result in a reduction of impurity—indicative of all nodes at that point representing the same class.

In addressing data imbalance, scikit-learn offers mechanisms to weight different samples through the *sample_weight* and *class_weight* parameters, which are designed to modify the sample weights and class weights respectively, thereby influencing the training outcome of the model (Buitinck et al., 2013). It is important to note that *sample_weight* and *class_weight* do not inherently correct data imbalance but can be utilized to mitigate its effects. These parameters function independently and may be used concurrently, though they target different aspects of the training process. The *sample_weight* parameter assigns a specific weight to each sample, provided as an array. This is particularly useful for emphasizing the importance of certain samples over others within the training dataset. When applying *sample_weight*, the model adjusts its training focus based on the weight of each sample, with prediction outcomes for each sample being weighted accordingly before averaging to produce the final result. Conversely, *class_weight* assigns weights to each class through a dictionary, aiming to balance the representation of different classes within the model. This parameter is especially valuable in classification tasks where certain classes are inherently more challenging to predict. Utilizing *class_weight* can enhance the model's sensitivity to these harder-to-predict classes. When set to "*balanced"*, *class_weight* automatically adjusts the weights to equalize the total sample weights across each class, facilitating a more equitable model training process.

Several strategies exist for modifying the sample weight through two key parameters. The initial approach involves designating the *class_weight* as "*balanced*". Delving into the *class_weight* parameter's underlying code reveals that with a "*balanced*" setting, the formula applied is: $\text{class\_weight} = \frac{n\_samples}{n\_classes \times np.bincount(y)}$, where *np.bincount(y)* tallies the frequency of each class (Amor and Liu, 2023). Absent an explicit *class_weight* setting, every class

default assumes a uniform weight of 1. An alternative method adjusts individual sample weights during the prediction model's fit function invocation, consequently scaling the computed loss by the pertinent *sample_weight* for the ultimate loss determination. This loss computation strategy ensures that the results are not disproportionately influenced by classes with fewer instances due to disparities. The *class_weight* and *sample_weight* parameters can be effectively utilized for combined weighting. By activating both parameters, the model integrates the impacts of sample and category weights, modifying how it trains. Initially, training is guided by the *sample_weight*; afterward, during the loss assessment phase, the error for each category is multiplied by its respective *class_weight* to compute the final loss function value. Through applying these diverse class weighting approaches, we discerned that the prediction model yields optimal performance when the *class_weight* is assigned "*balanced*", and *sample_weight* remains unspecified.

Regarding the *min_sample_leaf* parameter, pChanger employs the *GridSearchCV* utility to discern their optimal alignment, thereby optimizing the performance of the forecasting model. *GridSearchCV,* a tool within the *scikit-learn* library, streamlines the grid search process, automating the quest for the most effective hyperparameter amalgamation. It exhaustively explores every permutation of hyperparameters, ultimately presenting the premier mix as determined by cross-validation outcomes. This technique substantially alleviates the effort involved in hyperparameter tuning and furnishes a straightforward approach for enhancing model refinement.

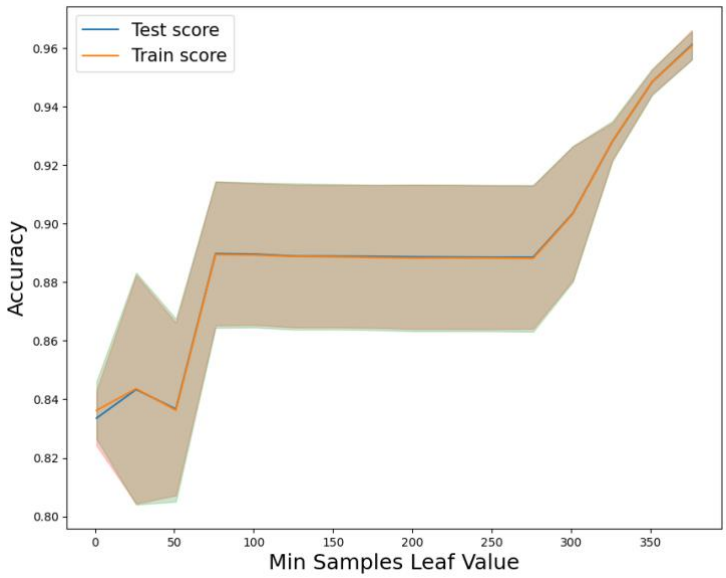

Figure 3-9 Training and Testing Score Trends with *min_samples_leaf* changing

The diagram labeled as Figure-3-9 above illustrates an example of the application of GridSearchCV in identifying the best setting for the *min_samples_leaf* parameter with parameter *slot_size*=2, ahead=4, heavy changer threshold ϕ=0.2, and package_threshold=20. It is evident from the graph that the standard deviation is considerably lower at min_samples_leaf=325. Beyond this point, the fluctuation in the standard deviation gets smaller and smaller as the min_samples_leaf value continues to rise. We use this method to find the appropriate min_samples_leaf parameter, balancing model performance and deployability on the data plane.

3.6 Model improvement

3.6.1 Multi-stage model training process

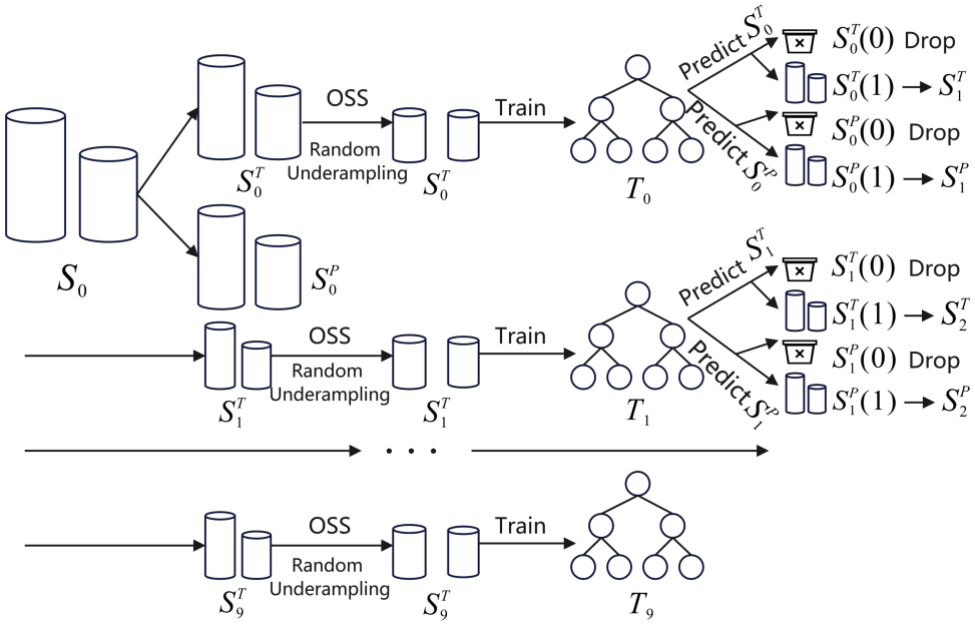

Figure 3-10 Multi-stage model training process

pChanger obtains an optimal model through multi-stage training. Initially, we split the dataset into a training set S_0^T and a validation set S_0^P. For the training set, we apply OSS (One-Sided Selection) and Random Undersampling to obtain the balanced training set S_0^T. We then train our first decision tree T_0 on S_0^T. This tree produces predictions for both the original training set and the validation set, resulting in four subsets: $S_0^T(0)$, $S_0^T(1)$, $S_0^P(0)$, and $S_0^P(1)$. We discard the non-heavy changer flows (i.e. $S_0^T(0)$ and $S_0^P(0)$) and retain the heavy changer flows as input for the next stage of training. By employing this method, the

accuracy of subsequent models has been enhanced. After 10 iterations, we obtain our final decision tree model T_9.

3.6.2 Post pruning

Pruning a decision tree doesn't rectify the skewed distribution of data; however, manual post-pruning is essential due to the excessive sample size that precludes the trained decision tree from being efficiently represented on the data plane.

Post-pruning in decision trees begins by eliminating specific leaf nodes from the tree's root, transforming their direct predecessors into new leaf nodes. This method assesses the tree's accuracy both prior to and following the pruning. Utilizing scikit-learn, this approach modifies the *ccp_alpha* parameter to implement Cost Complexity Pruning, where *ccp_alpha*, a positive value, measures the balance between node intricacy and training mistakes, thus determining the level of pruning (Buitinck et al., 2013).

During the cost complexity reduction, a higher regularization parameter indicated by *ccp_alpha* correlates with more aggressive trimming of the tree; conversely, a lower *ccp_alpha* suggests that the resultant subtree will more closely resemble the initial comprehensive decision tree. Table 3-2 illustrates the specific steps involved in cost complexity pruning.

Table 3-2 Steps of Cost Complexity Pruning

Step 1	Compute the *ccp_alpha* value for each node in an ascending order, starting from the lowest node.
Step 2	Start with the full decision tree and generate a sequence of subtrees by progressively applying different *ccp_alpha* threshold values. Each subtree in this series corresponds to a distinct *ccp_alpha* value.
Step 3	Using cross-validation across these subtrees, the superior subtree is chosen as the final model by assessing whether it attains the highest mean accuracy (for classification tasks) or the greatest mean variance (for regression tasks).

In Figure 3-11, the accuracy and F1-score of both the training and test sets are presented, employing a *slot_size* of 5, ϕ=0.2, and *ahead*=1, as the *ccp_alpha* value undergoes variations. pChanger modifies the *ccp_alpha* to evaluate the pruned decision tree's size, aiming to

maintain a complex model within the allowable limits of the data plane by selecting the minimal viable *ccp_alpha*. The evaluation metric used is the F1 score, with multiple training iterations conducted and their average values reported.

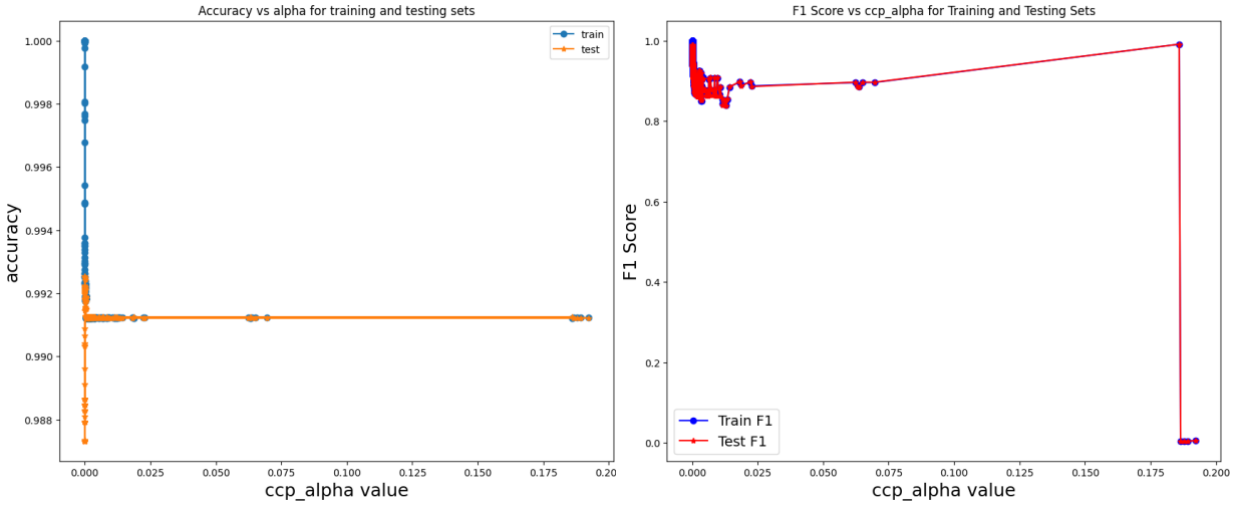

Figure 3-11 Accuracy (a) and F1-score (b) of Training and Testing Sets Against *ccp_alpha*

Following these adjustments and the pruning process, pChanger finalizes its prediction model. The final model we selected uses *slot_size*=2, *package_threshold*=20, *ahead*=4, and heavy changer threshold ϕ=0.2. To reduce the complexity of the decision tree, we combinde the will-change flow and heavy changer categories into a single positive category, while keeping the normal flow as negative. This model was trained and refined using the methods described in Sections 3.5 and 3.6.

3.7 Data plane design

3.7.1 Challenges and solutions in data plane implementation

The implementation of pChanger utilizes the P4 programming language specifically designed for packet driving. This framework restricts pChanger to only accessing the metadata of the package presently incoming pipeline. Consequently, it cannot predict or access features of other network flows based on the information from a single packet. This limitation presents a significant challenge: extracting and utilizing network traffic features over time within the constraints of packet-based data flow in the data plane.

Furthermore, pChanger encounters a critical balancing act in the data plane between the accuracy of predictions and the speed of processing. The extent of flow information collected before initiating flow identification greatly influences this balance. For instance, in scenarios where the switch conducts load balancing, a small number of false negatives might be tolerable, emphasizing the need for speed over precision. Conversely, in situations where heavy flows are indicative of potential denial-of-service attacks, the accuracy of identifying such threats is crucial to prevent false alarms that could burden network operators. Given the two mentioned challenges and switch resource constraints, a strategy is necessary to effectively orchestrate the operational logic of pChanger across various stages.

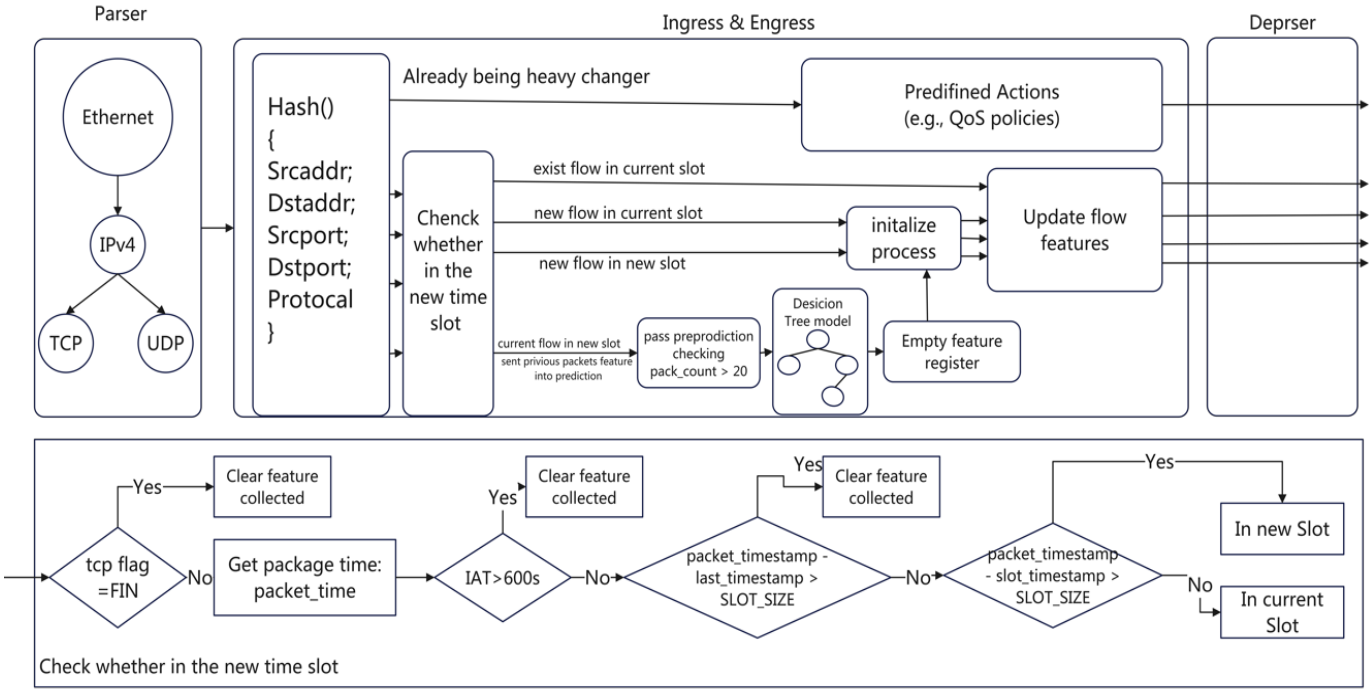

Figure 3-12 pChanger Pipeline Overview

In Figure 3-12, the pipeline architecture of pChanger is detailed within the programmable data plane. Packets are first processed in the Parser where headers are dissected to identify flow characteristics such as TCP flags and packet inter-arrival times (IAT). Following this, the packets and processed data advance through the pipeline. During their journey through both the Ingress and Egress pipeline, packets are organized into separate flows according to their five-tuple data, which includes both source and destination IP addresses, ports, and protocol numbers. pChanger leverages this five-tuple information in packet headers to uniquely associate packets with flows through a hashing algorithm facilitated by the P4 language.

Moreover, the P4 switch assigns timestamps at the ingress point, enabling the calculation of time-sensitive variables like IAT. These attributes are maintained in registers and manipulated using basic operations that P4 supports, such as addition, subtraction, and hashing. For instance, if a packet arrives with the ACK flag set, the respective counter for the ACK flag is incremented. Due to the absence of division operations in P4, pChanger implements an exponentially weighted moving average (EWMA) for its averaging tasks (Busse-Grawitz et al., 2022). EWMA is characterized by the following formula:

$$S_t = \begin{cases} y_1 & t = 1 \\ \alpha \cdot y_t + (1 - \alpha) \cdot S_{t-1} & t > 1 \end{cases}$$

Adjusting the α parameter to 0.5 allows the switch to perform averaging calculations using bitwise methods. Before exiting the system, all packets are processed through an inverse resolution stage (Deparser) to ensure they are correctly reassembled.

3.7.2 Time-slot based feature prediction in P4 programming

Given the P4 language's packet-driven nature, it can capture only the hash value for the flow associated with the currently processed packet, enabling predictions based on features accumulated from the last slot for that specific flow using a decision tree approach. However, acquiring features of other flows via the current packet is not feasible, preventing the prediction of a comprehensive feature set from previous slots.

To address this issue, pChanger uses three temporal registers: *last_timestamp* to record the timestamp of the last packet for each flow, *cur_slot* to indicate the current slot number, and *slot_timestamp* to store the timestamp of the first packet in the current slot. The process for determining whether a new slot has begun involves comparing the timestamp of the current packet (defined as *packet_time*) and the last packet (defined as *last_timestamp_temp*) with the timestamp of the first packet in the current slot (defined as *start_time*). If the current packet time difference exceeds a predefined slot size (defined as *SLOT_SIZE*), a new slot is initiated, and the features from the previous slot are stored in temporary variables while the registers are cleared. This process resets the *start_time* to the current packet's timestamp and updates the *slot_timestamp* and *cur_slot* registers accordingly. The features from the previous slot are used for heavy changer prediction then the feature registers are cleared. This mechanism ensures that each flow is accurately tracked and managed within its respective time slot. It is crucial to clear the features collected and stored in various registers after each

prediction round which ensures that subsequent predictions for the flow are not adversely influenced by residual data from previous rounds.

Illustrated by Figure 3-2 and postulating a *SLOT_SIZE* of 2 seconds, we assume that each slot shown in the figure also lasts for 2 seconds. We can observe that the first packet to arrive is from Flow A, at which point the timestamp of this packet is stored in the *slot_timestamp* register, and *cur_slot* is initialized to 1. Subsequently, the switch receives many packets from both Flow A and Flow C. As seen in the figure, a segment of Flow C's packet transmission spans Slot 1 and Slot 2. Therefore, when the switch reads a packet from this stage, it will find one package from Flow C with the difference between the packet's timestamp (*packet_time*) and the *start_time* exceeds the *SLOT_SIZE*. At this point, the value in the *cur_slot* register is incremented by one, and the *slot_timestamp* is updated to the timestamp of this packet. The model then exports the statistical data of Flow C from the registers, inputs it into the decision tree for prediction to obtain the prediction result, clears the feature registers, and starts collecting flow features for the new time slot. The processing logic for Flow A during the transition from Slot 2 to Slot 3 is the same as Flow C from Slot 1 to Slot 2.

The prediction for Flow A in Slot 1 is initiated when the first packet in Slot 2 arrives, and the processing logic is consistent with the handling of Flow C during the transition from Slot 2 to Slot 3, as described below. For Flow C during the transition from Slot 2 to Slot 3, when the first packet of Slot 3 arrives, the model determines that *package_time* is greater than *start_time*, and at the same time, *last_timestamp_temp* is less than *start_time*. At this point, the features of Flow C are directly fed into the decision tree for prediction. Flow B has traffic only in Slot 2, so Flow B will not be used in the decision tree prediction.

It is worth noting that the update from Slot 2 to Slot 3 may be triggered by a packet from Flow A or Flow D, depending on which flow's packet first causes the difference between the current timestamp and *slot_timestamp* to exceed *SLOT_SIZE*.

3.7.3 Prediction model

Two preliminary checks are essential when the prediction module starts. The first step involves verifying if the *packet_count*, collected from the features of the current flow in the last slot, falls under 20. Should this be the case, the pChanger system will automatically exclude these features. This approach ensures consistency with the feature sample data

post-cleansing before model training, while also accelerating the model's prediction speed. Subsequently, the module employs the *flow_hash* to retrieve the status flag for each flow from the *flag_reg*. A flag value of 1 indicates that the flow has already been predicted, whereas a flag value of 0 suggests that the prediction is still pending. pChanger uses *res_flag* to mark the prediction results; if a flow is predicted to be a heavy changer, *res_flag* is set to 1. Due to the decision tree's *ahead*=4, a flow predicted as a heavy changer might currently be a heavy changer or become one within the next 1 to 4 slots.

A fundamental challenge in online classification, as identified by Busse-Grawitz et al (2022), is determining the appropriate timing for classification. In the pChanger system, the collection period for flow features is predetermined as a slot time, which shifts the challenge to deciding how many predictions should be made on the same flow to accurately determine its final status. Premature predictions can result in insufficient data, leading to the unreliable classification of flows, particularly in scenarios demanding high prediction accuracy for heavy changers, such as in suspected denial-of-service attacks. Conversely, aiming for higher prediction accuracy can lead to issues when predictions are delayed; for example, in the context of Pulse-Wave attacks, delayed classification could cause service disruptions. Additionally, late classification consumes excessive computational resources by maintaining and storing flow features longer than necessary. It is crucial to note that the balance between accuracy and prediction timing can vary significantly depending on the network environment.

pChanger introduces the As Soon As Possible (ASAP) model which is deployed when rapid identification of large flows is critical. In the ASAP model, prediction for a flow ceases once it is identified as a heavy changer. This is implemented by setting the identification field in the IP header to 1 if the prediction result (*res_flag*) for that flow is 1. Finally, when a flow receives a termination signal with a flag value of 1 (i.e., *terminal_flag*=1), pChanger efficiently frees up the associated memory within the programmable data plane, stopping further predictions and resource allocation for that flow.

4. Evaluation

4.1 Experiement setup

1) Testbed: We conduct our experiments on a virtual server equipped with 4 cores of Intel

i7-8550U 1.80GHz CPU and 20GB RAM. The Sever runs Ubuntu 20.04.6 LTS. To exclude the I/O overhead on performance, we load all datasets into memory before all experiments. We utilize the simple switch in bmv2 as the experimental platform for the data plane experiments, which runs directly on our virtual server. It is initiated using the setup.sh script provided by bmv2, with veth2 as the input interface and veth4 as the output interface.

2) Dataset: We use an open-source dataset from one of the ten data centers called UNIV1 studied in the IMC 2010 paper titled Network Traffic Characteristics of Data Centers in the Wild (Benson, Akella and Maltz, 2010). The payload has been nulled out and the IP addresses have been anonymized using SHA1 hash to anonymize the data. We divided the traces into some time slots based on different *slot_size*, *slot_size* is from 2 to 5 seconds. For the experiments on the P4 programmable data plane, we used the first 6 data packets obtained by dividing UNIV1 with a *slot_size* of 2 seconds, resulting in a dataset of 29,393 packets.

3) Methodology: For evaluation purposes, we generate the ground truths by finding D and hence mark true heavy changer flows for different slots based on heavy changer threshold $\phi = 0.2$ and *slot_size=2* as the length of epochs. Since there is no research on the prediction aspect of heavy changers, we compare it here with some state-of-the-art sketch-based detection methods on the data plane, typically MV-Sketch (Tang et al., 2019).

We used two different methods to limit memory usage directly in the experiments on the dataset. We first stored the data in memory. We followed the original authors' method for sketch-based methods by maintaining the sketch depth (i.e., the number of rows) and varying the number of buckets per row to impose limits. The number of buckets depends on how they are designed. For MV-Sketch, a bucket includes an 8-byte flow key and two 4-byte counters. We obtained the sketch width (i.e., the number of buckets per row) by dividing the total memory allocated by the sketch depth and bucket size. We limited the memory used by sketch methods by changing the sketch width. For our pChanger model, we directly used the system's built-in *resource.setrlimit* to limit runtime memory usage. Where the original works did not specify settings, we referred directly to the default configurations in the provided source codes. For hash functions, we adopted MurmurHash, following the settings used in the referenced studies. We utilized the source-destination address pair (64 bits) as the flow key for MV-Sketch (Tang et al., 2019). For the p4 implementation of the decision tree, we follow the

statement in section 3.7.3. We first complete the feature extraction phase. For the average calculations used in feature extraction, we employ EWMA. Next, we perform downward rounding for the floating-point numbers involved in the original decision tree to obtain our decision tree model in the data plane.

4) Metrics: We consider the following metrics.

Accuracy: proportion of correctly identified flows, both heavy changers and non-heavy changers, among all evaluated flows;

Precision: fraction of true heavy changer reported overall reported flows;

Recall: fraction of true heavy flows reported. over true heavy changer flows.

F1-score: $\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$

Relative error: $\frac{1}{|R|}\sum_{x,x\in R}\frac{|S(x)-S(x)|}{S(x)}$, where R is the set of true heavy flows reported.

Update throughput: number of packets processed per second (in units of Mbits/s).

4.2 Results and analysis

We compare pChanger with one state-of-art detection method MV-Sketch (Tang et al., 2019), the results are shown below.

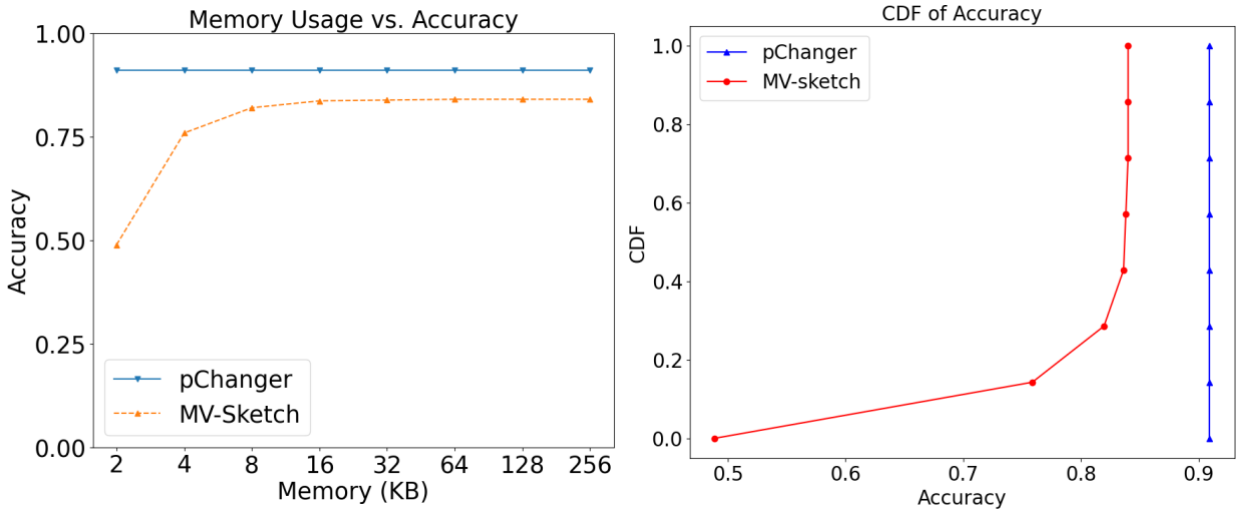

Figure 4-1 Experiment 1 (Accuracy for heavy changer prediction and detection)

Figure 4-1 compares the accuracy of pChanger and MV-Sketch methods in the context of heavy changer detection across varying memory capacities. MV-Sketch shows significantly improved accuracy as memory utilization increased from 2KB to 4KB, with an approximate 27% increase. This increase suggests that the Sketch-based method benefits significantly from

the initial allocation of additional memory, that due to the increase in its bucket number, which makes it detect different flows more exactly. Beyond the 4KB threshold, the accuracy of MV-Sketch plateaus, suggesting that MV-Sketch reaches an optimal point where additional memory does not help a lot to improve MV-Sketch in being able to better detect changes in traffic accurately. pChanger demonstrates a consistently high accuracy across all tested memory capacities, starting from 2KB up to 256KB and outperforming MV-Sketch. This stability shows that pChanger is inherently efficient in small resource consumption to achieve a high level of accuracy with minimum memory. Therefore, pChanger is especially useful in memory constrained setups.

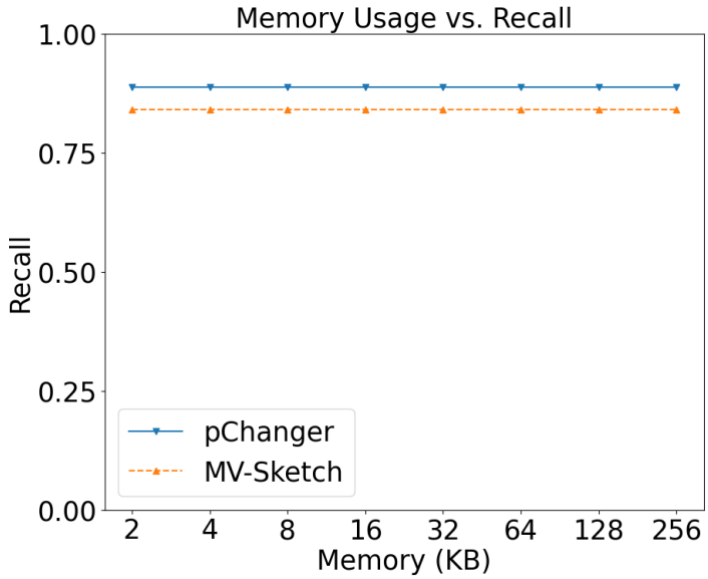

Figure 4-2 Experiment 2 (Recall for heavy changer prediction and detection)

Figure 4-2 shows the recall for heavy changer prediction and detection, among these two methods, the MV-Sketch's recall increased from 0.81 using 2KB memory to 0.84 using 8KB memory and remained unchanged after that. Both methods start with a relatively high recall rate, indicating strong performance even at lower memory capacities. pChanger maintains a steady recall rate of approximately 0.89 across all tested memory sizes from 2KB to 256 KB. On the other hand, MV-Sketch shows a slight improvement as the memory allocation is increased. Starting from a recall of 0.81 at 2KB, it experiences a notable increase to 0.84 when the memory is expanded to 8KB. Beyond this point, the recall of MV-Sketch plateaus, remains stable at around 0.84 through to 256KB of memory. This trend is consistent with previous experiments, where MV-Sketch gets a performance boost due to additional memory

improvements, while pChanger is less dependent on memory and outperforms MV-Sketch.

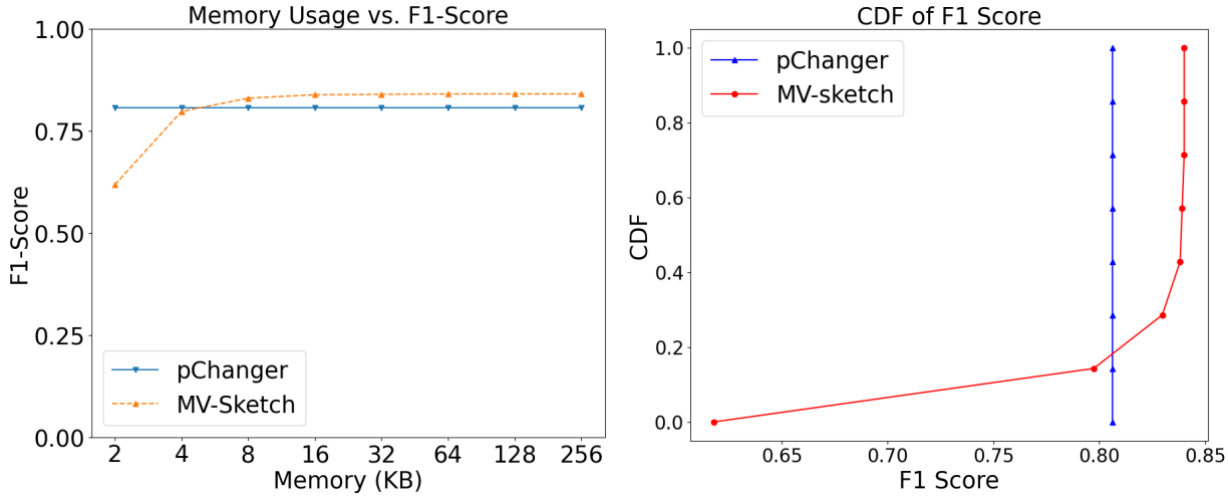

Figure 4-3 Experiment 3 (F1-Score for heavy changer prediction and detection)

Figure 4-3 illustrates the relationship between memory usage and the F1-score for detecting heavy changers in network traffic, comparing two methodologies: pChanger and MV-Sketch. As observed, the pChanger method demonstrates better performance at lower memory capacities. With just 2KB of memory, pChanger achieves an F1-score approximately above 0.75, slightly better than MV-Sketch, which scores just below this mark. This trend continues as memory increases to 4KB, maintaining a close competition. However, the performance dynamics change as more memory is allocated. Starting from 8KB and extending to 256KB, MV-Sketch shows a consistent F1-score slightly above 0.75. In contrast, pChanger's F1-score converges with that of MV-Sketch at 8KB and remains constant, mirroring MV-Sketch's performance across the higher memory capacities.

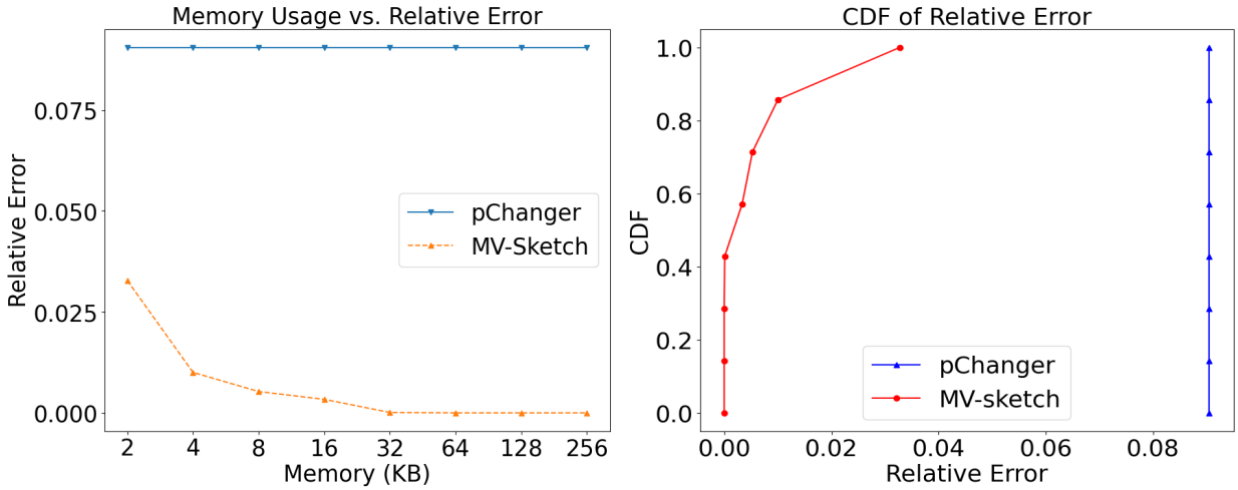

Figure 4-4 Experiment 4 (Relative error for heavy changer prediction and detection)

Figure 4-4 depicts the relationship between memory usage (KB) and relative error for two algorithms: pChanger and MV-Sketch. pChanger, remaining constant at approximately 0.09 across all memory usages. Conversely, the curve for MV-Sketch demonstrates a significant decrease in relative error as memory usage increases, achieving near-zero relative error beyond 32KB. This stark difference can be attributed to MV-Sketch's ability to leverage additional memory to store more detailed and accurate summaries of data, thereby enhancing accuracy. The higher relative error of pChanger compared to MV-Sketch is due to the model's inability to fully utilize the additional memory; we can improve this by incorporating multiple trees for prediction while staying within memory constraints.

Table 4-1 pChanger performance on programmable data plane

Method	**Accuracy**	**Recall**	**Precision**	**F1-Score**
pChanger	0.9997	0.7500	0.6000	0.6667

As mentioned in Section 4.1, we used the first 6 slots of data with a *slot_size* of 2 seconds for testing. The model predicted that 12 packets from 2 flows were heavy changers, while the actual value marked 16 packets as heavy changers. This gave us an initial performance evaluation of the model on the P4 switch. The reason for the high accuracy might be due to the small number of packets; as the number of packets increases, the accuracy is expected to decrease. We observed that rounding floating-point numbers to integers in the decision tree reduces model performance. However, our model can predict heavy changers up to 4 slots in advance on the programmable data plane, allowing for proactive monitoring and actions. Compared to the detection method, this reduces the probability of unnecessary network congestion.

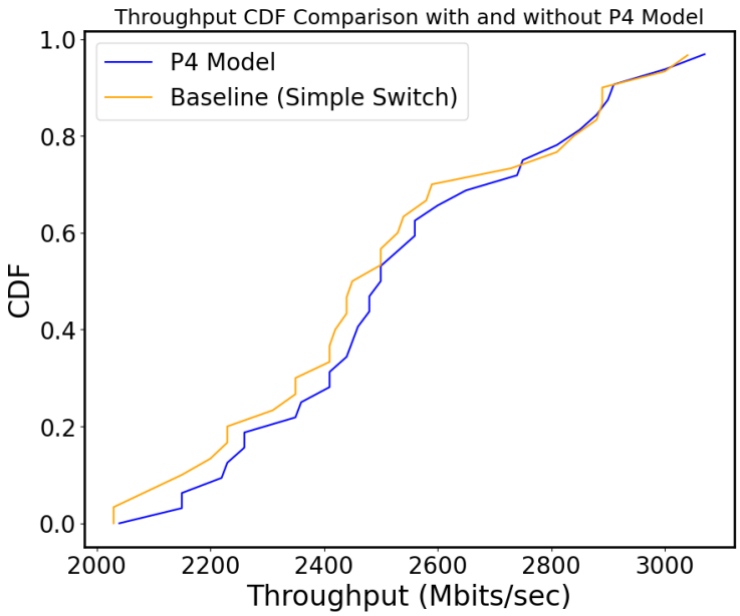

Figure 4-5 Experiment 6 throughput comparison of simple switch with and without pChanger

To evaluate the performance of pChanger on a P4 software switch, we used *iperf* to send UDP packets with a 10G sending bandwidth for 30 seconds to both a Simple Switch with the pChanger model loaded and a normal Simple Switch, reporting results every second. As shown in Figure 4-5, when measured in Mbits/sec, we can observe that the switch with the model is slightly slower. However, its average throughput is 2.5 Gbits/sec, which is on par with the Simple Switch without the model. This demonstrates that pChanger can predict at line rate. After incorporating the model, the packet loss rate remained at a relatively low level.

5. Conclusion

In this paper, we introduce pChanger, a decision-tree-based system that enables programmable data planes to predict heavy changers based on time slots. Compared to traditional methods, it can predict and monitor data flows before they become heavy changers, thereby significantly minimizing network congestion caused by such changers. The constraint of time slots facilitates the controller to process the data based on the current time point, enhancing timeliness. pChanger employs decision tree algorithms for prediction and has demonstrated through dataset usage that it can achieve higher accuracy and recall with just one time slot's information, despite still having a lower F1 score and issues with false positives compared to traditional methods. Experiments in P4 have also confirmed pChanger's

high accuracy.

However, decision-tree-based machine learning methods still struggle to adapt to complex and variable network environments. The first limitation is that the model's performance is too rigid; although it performs well with low memory, it cannot utilize additional memory when available. Future directions might explore using multiple decision trees or a random forest to improve this issue, such as introducing more trees when additional memory is available or allowing controllers to dynamically configure and modify machine learning models. The model's generalization capability is also a concern; our performance on an unseen dataset UNIV2 was very poor. Combining the UNIV1 and UNIV2 datasets yielded a similar but slightly improved performance as we observed. To enhance performance, this might require trying more innovative and convertible machine learning models for P4, such as gradient boosting trees.

Acknowledgments

As my university days draw to a close, I want to take this opportunity to express my heartfelt gratitude to all those who have helped me along the way.

First and foremost, I would like to thank my thesis advisor, Professor Cui Lin. His high level of expertise, rigorous standards, and tireless dedication to teaching have been invaluable to me. His approachable demeanor and personal charm have left a profound impact on me, and I am confident that his influence will continue to shape my future endeavors. His support was indispensable from the selection of my thesis topic to its completion. I am also deeply grateful to the students and teachers at the International School. It is hard to imagine encountering such a group of dedicated educators. They have provided us with an education comparable to that of top-tier universities worldwide and have helped me tremendously with career planning and revising my application documents. The unforgettable moments shared with my classmates are precious, and I hope to stay in touch with them always and wish them a bright future. Furthermore, I extend my thanks to the many senior students who lent a helping hand when I needed it most. I am also thankful for some of my high school friends and university peers whose comfort and encouragement were crucial during tough times. I owe a special thanks to the virtual idols of ASOUL. Their presence offered me solace and motivation during moments of uncertainty and despair, reigniting my drive to move forward. I am also grateful to Japanese anime and football for the excitement and relaxation they have provided, which helped me unwind and recharge. Most importantly, I am grateful to my parents. Without their companionship, understanding, and support, I would not be where I am today. Their unwavering faith in me has given me the courage and confidence to make bold choices. Finally, I would like to thank myself for successfully transitioning from biotechnology to computer science and technology, and for graduating successfully.

References

Amor, A., & Liu, L. (2023). Scikit-learn(v.1.3.2) documentation. Retrieved from: https://scikit-learn.org/1.3/user_guide.html. (accessed 19 April 2024)

Benson, T., Akella, A., & Maltz, D. A. (2010). Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (pp.267–280). Melbourne: Association for Computing Machinery.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., & Walker, D. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 87–95.

Breiman, L., Friedman, J., Olshen, R. A., & Stone, C. J. (2017). Introduction to tree classification. In L. Breiman, J. Friedman, R. A. Olshen, & C. J. Stone, Classification and Regression Trees (pp. 18-58). New York, State of New York: Chapman and Hall/CRC.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., Vanderplas, J., Joly, A., Holt, B., & Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. Retrieved from: https://doi.org/10.48550/arXiv.1309.0238. (accessed 20 April 2024)

Busse-Grawitz, C., Meier, R., Dietmüller, A., Bühler, T., & Vanbever, L. (2022). pForest: In-Network Inference with Random Forests. Retrieved from: https://doi.org/10.48550/arXiv.1909.05680. (accessed 21 April 2024)

Carvalho, R. N., Costa, L. R., Bordim, J. L., & Alchieri, E. A. P. (2021). Detecting DDoS Attacks on SDN Data Plane with Machine Learning. In 2021 Ninth International Symposium on Computing and Networking Workshops (pp.138–144). Matsue: IEEE Computer Society.

Chen, X., Landau-Feibish, S., Braverman, M., & Rexford, J. (2020). BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. Retrieved from: https://doi.org/10.1145/3387514.3405865. (accessed 24 April 2024)

Cormode, G., & Muthukrishnan, S. (2005). An improved data stream summary: The

count-min sketch and its applications. *Journal of Algorithms*, 55(1), 58–75.

Deepa, V., Sudar, K. M., & Deepalakshmi, P. (2018). Detection of DDoS Attack on SDN Control plane using Hybrid Machine Learning Techniques. In 2018 International Conference on Smart Systems and Inventive Technology (pp.299–303). Tirunelveli: Francis Xavier Engineering College.

Durand, M., & Flajolet, P. (2003). Loglog Counting of Large Cardinalities. Retrieved from: https://link.springer.com/chapter/10.1007/978-3-540-39658-1_55. (accessed 19 April 2024)

Gebara, N., Lerner, A., Yang, M., Yu, M., Costa, P., & Ghobadi, M. (2020). Challenging the Stateless Quo of Programmable Switches. Retrieved from: https://doi.org/10.1145/3422604.3425928. (accessed 22 January 2024)

Hart, P. E. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14(3), 515–516.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., & Menth, M. (2023). A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212, 103561.

He, M., Basta, A., Blenk, A., Deric, N., & Kellerer, W. (2018). P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration. In 2018 14th International Conference on Network and Service Management (pp.90–98). Rome: Consorzio Nazionale Interuniversitario per le Telecomunicazioni.

Huang, J., Zhang, W., Li, Y., Li, L., Li, Z., Ye, J., & Wang, J. (2023). ChainSketch: An Efficient and Accurate Sketch for Heavy Flow Detection. *IEEE/ACM Transactions on Networking*, 31(2), 738–753.

Huang, N.-F., Jai, G.-Y., Chao, H.-C., Tzang, Y.-J., & Chang, H.-Y. (2013). Application traffic classification at the early stage by characterizing application rounds. *Information Sciences*, 232, 130–142.

Huang, Q., & Lee, P. P. C. (2014). LD-Sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In The 33rd Annual IEEE International Conference on Computer Communications (pp.1420–1428). Toronto: IEEE

Computer Society.

Huang, Q., Lee, P. P. C., & Bao, Y. (2018). Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. Retrieved from: https://dl.acm.org/doi/10.1145/3230543.3230559. (accessed 20 March 2024)

IANA. (2024). Protocol Numbers. Retrieved from: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml. (accessed 14 April 2024)

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

Kubat, M., & Matwin, S. (1997). Addressing the curse of imbalanced training sets: One-sided selection. *International Conference on Machine Learning*, 97(1), 179.

Li, W., & Patras, P. (2023). Tight-Sketch: A High-Performance Sketch for Heavy Item-Oriented Data Stream Mining with Limited Memory Size. Retrieved from: https://dl.acm.org/doi/10.1145/3583780.3615080. (accessed 10 April 2024)

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., & Turner, J. (2008). OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 69–74.

Nguyen, T. T. T., & Armitage, G. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4), 56–76.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85), 2825–2830.

Poupart, P., Chen, Z., Jaini, P., Fung, F., Susanto, H., Geng, Y., Chen, L., Chen, K., & Jin, H. (2016). Online flow size prediction for improved network routing. In 2016 IEEE 24th International Conference on Network Protocols (pp.1–6). Singapore: IEEE Computer Society.

Tang, L., Huang, Q., & Lee, P. P. C. (2019). MV-Sketch: A Fast and Compact Invertible

Sketch for Heavy Flow Detection in Network Data Streams. In 2019 IEEE International Conference on Computer Communications (pp.2026–2034). Paris: IEEE Communications Society.

The imbalanced-learn developers. (2024). RandomUnderSampler. Retrieved from: https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html. (accessed 10 April 2024)

Tomek, ivan. (1976). Two Modifications of CNN. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11), 769–772.

Xavier, B. M., Guimaraes, R. S., Comarela, G., & Martinello, M. (2021). Programmable Switches for in-Networking Classification. Retrieved from: https://ieeexplore.ieee.org/document/9488840. (accessed 22 April 2024)

Xiong, Z., & Zilberman, N. (2019). Do Switches Dream of Machine Learning? Toward In-Network Classification. Retrieved from: https://doi.org/10.1145/3365609.3365864. (accessed 19 April 2024)

Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., & Uhlig, S. (2018). Elastic sketch: Adaptive and fast network-wide measurements. Retrieved from: https://dl.acm.org/doi/10.1145/3230543.3230544. (accessed 5 April 2024)

Yu, M., Jose, L., & Miao, R. (2013). Software Defined Traffic Measurement with OpenSketch. Retrieved from: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu. (accessed 28 March 2024)

Zheng, H., Jiang, Y., Tian, C., Cheng, L., Huang, Q., Li, W., Wang, Y., Huang, Q., Zheng, J., Xia, R., Wang, Y., Dou, W., & Chen, G. (2022). Rethinking Fine-Grained Measurement From Software-Defined Perspective: A Survey. *IEEE Transactions on Services Computing*, 15(6), 3649–3667.