This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



# NELLY: Flow Detection Using Incremental Learning at the Server-Side of SDN-based Data Centers

Felipe Estrada-Solano, *Graduate Student Member, IEEE,* Oscar M. Caicedo, *Member, IEEE,* and Nelson L. S. da Fonseca, *Senior Member, IEEE,*

***Abstract*—The processing of big data generated by the Industrial Internet of Things (IIoT) calls for the support of processing at the edge of the network as well as at the cloud data centers. The Equal-Cost Multi-Path (ECMP), which is the default routing technique in the cloud data centers, can degrade the network performance when handling mouse and elephant flows. Such degradation of performance can compromise the support of the strict quality of service requirements of IIoT over 5G networks. Novel techniques for scheduling the elephant flows can alleviate this problem. Recently, several approaches have incorporated Machine Learning (ML) techniques at the controller-side in Software-Defined Data Center Networks (SDDCNs) to detect elephant flows. However, these approaches can produce heavy traffic overhead, low scalability, low accuracy, and high detection time. This paper introduces the Network Elephants Learner and anaLYzer (NELLY), a novel and efficient method for applying incremental learning at the server-side of SDDCNs to accurately and timely identify elephant flows with low traffic overhead. Incremental learning enables NELLY to adapt to varying network traffic conditions and perform continuous learning with limited memory resources. NELLY has been extensively evaluated using real traces and various incremental learning algorithms. Results show that NELLY is accurate and supports low classification time when using adaptive decision trees algorithms.**

***Index Terms*—Data center networks, flow classification, machine learning, software defined networking**

## I. Introduction

The Industrial Internet of Things (IIoT) aims at automating industrial processes that can be supported by the analysis of big data generated by a large number of interconnected devices [1]. The real-time requirements of industrial applications and the access demand of massive machine-type communications call for the employment of the 5G technology in the foreseen IIoT. In industrial plants, edge devices (fog nodes) will be used for the processing of delay-sensitive data while cloud servers will be employed for the processing of the huge amount of data generated by sensors. Extracting value from such big data will be fundamental for enabling customized and flexible mass production of goods [2], [3].

To support the big data demand in IIoT, cloud data centers provide significant bandwidth capacity for a large number of servers interconnected by a specially designed network, called Data Center Network (DCN) [4]. This bandwidth capacity can be optimized by using *multipath routing*, which distributes traffic over multiple concurrent paths [5]. Nowadays, the Equal-Cost Multi-Path (ECMP) is the default multipath routing mechanism for DCNs [6]. However, ECMP can degrade the performance of DCNs due to the coexistence of many small, short-lived flows (i.e., mice) and few large, long-lived flows (i.e., elephants), since ECMP can assign more elephant flows to the same path, generating *hot-spots* (i.e., some links overused while others underused). Flows traversing hot-spots suffer from low throughput and high latency. Mice and elephants are characteristic in cloud data centers running Big Data Analytics (BDA) technologies [7], [8], [9], such as MapReduce and Hadoop, which are pivotal in IIoT systems for delivering value from the big data and making business decisions [10]. Then, similar flows will be present in the cloud data centers of IIoT systems. As a consequence, DCNs that use only ECMP for managing the bandwidth demanded by the big data in IIoT likely will not meet the target efficiency specified for 5G networks.

Recent multipath routing mechanisms have leveraged Software-Defined Networking (SDN) to face the ECMP limitations; DCNs using SDN are referred to as Software-Defined Data Center Networks (SDDCNs). SDN allows a logically centralized controller to dynamically make and install routing decisions on the basis of a global view of the network [11], [12]. SDN-based multipath routing dynamically reschedules elephant flows, while handling mouse flows by employing default routing such as ECMP [6] and MiceDCER [13]. Reactive *flow detection* methods, which are at the heart of SDN-based mechanisms, discriminate elephants from mice by using static thresholds [14], [15], [16]. However, reactive methods are not suitable for SDDCNs since hot-spots may occur before the elephant flows are detected.

Novel SDN-based flow detection methods incorporate Machine Learning (ML) for proactively identifying elephant flows [17]. However, ML-based methods operate at the *controller-side* of SDDCNs, requiring the central collection of either *per-flow* data [18] or *sampling-based* data [19], [20]. The

Manuscript received May 30, 2019; revised September 9, 2019; accepted October 7, 2019. This work was supported in part by the Administrative Department of Science, Technology and Innovation of Colombia (COLCIENCIAS) under Grant 647-2014 and in part by the São Paulo Research Foundation (FAPESP), Brazil, under Grant #2015/24494-8 and Grant #2019/04914-3. Paper no. TII-19-2157. *(Corresponding author: Nelson L. S. da Fonseca.)*

F. Estrada-Solano is with the Department of Telematics, University of Cauca, Popayan 190003 Colombia, and also with the Institute of Computing, State University of Campinas, Campinas, SP 13083-852 Brazil (email: festradasolano@unicauca.edu.co).

O. M. Caicedo is with the Department of Telematics, University of Cauca, Popayan 190003 Colombia (email: omcaicedo@unicauca.edu.co).

N. L. S. da Fonseca is with the Institute of Computing, State University of Campinas, Campinas, SP 13083-852 Brazil (email: nfonseca@ic.unicamp.br).

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



central collection of per-flow data, however, causes problems such as heavy traffic overhead and poor scalability. Sampling-based data, on the other hand, tends to provide delayed and inaccurate flow information. Moreover, sampling techniques that mitigate the problem rely on non-standard SDN specifications. Using ML on either the *switch-side* or *server-side* represents a potential solution to the controller-side problems since these locations enable prompt and per-flow data with low traffic overhead. Switch-side flow detection methods based on ML are impractical because they require specialized hardware and put a heavy processing load on the switches. Conversely, ML-based flow detection methods at the server-side require only software modifications in the servers; nonetheless, these methods have not been fully explored.

In this paper, we propose a novel flow detection method denominated Network Elephants Learner and anaLYzer (NELLY), which applies incremental learning at the server-side of SDDCNs for accurately and timely identifying elephant flows while generating low control overhead. Incremental learning allows NELLY to constantly train a flow size classification model from continuous and dynamic data streams (i.e., flows) [17], [21], providing a constantly updated model and reducing time and memory requirements. Thus, NELLY adapts to the variations in traffic characteristics and performs endless learning with limited memory resources. We extensively evaluate NELLY using datasets extracted from real packet traces and incremental learning algorithms. Quantitative evaluation demonstrates that NELLY is efficient in relation to accuracy and classification time when adaptive decision trees algorithms are used. Analytic evaluation corroborates that NELLY is scalable, causes low traffic overhead, and reduces detection time, yet it is in conformance with SDN standards.

The remainder of this paper is as follows. Section II introduces NELLY. Section III presents a quantitative evaluation of NELLY using incremental learning algorithms and real packet traces. Section IV compares NELLY to other related work. Section V concludes the paper.

## II. NELLY

Fig. 1 introduces NELLY, a flow detection method that applies incremental learning at the server-side of SDDCNs to identify elephant flows accurately in a reasonable time while generating low control overhead. NELLY operates as a software component either in the kernel of the host Operating System (OS) or in the hypervisor of servers in the SDDCN with the aim of monitoring all packets sent by the applications, containers, and virtual machines. Since NELLY detects elephant flows at their origin, a small overhead is demanded.

The architecture of NELLY (see Fig. 1) has two subsystems: *Analyzer* and *Learner*. The Analyzer applies a flow size classification model for detecting and marking elephant flows on the fly. The Learner then applies an incremental learning algorithm for building and updating the flow size classification model. This model maps online features (i.e., features extracted from the first few packets of a flow) onto the corresponding class of flows (i.e., mice or elephants). The processes of the Analyzer and the Learner run concurrently as depicted in Algorithms 1 and 2, respectively.

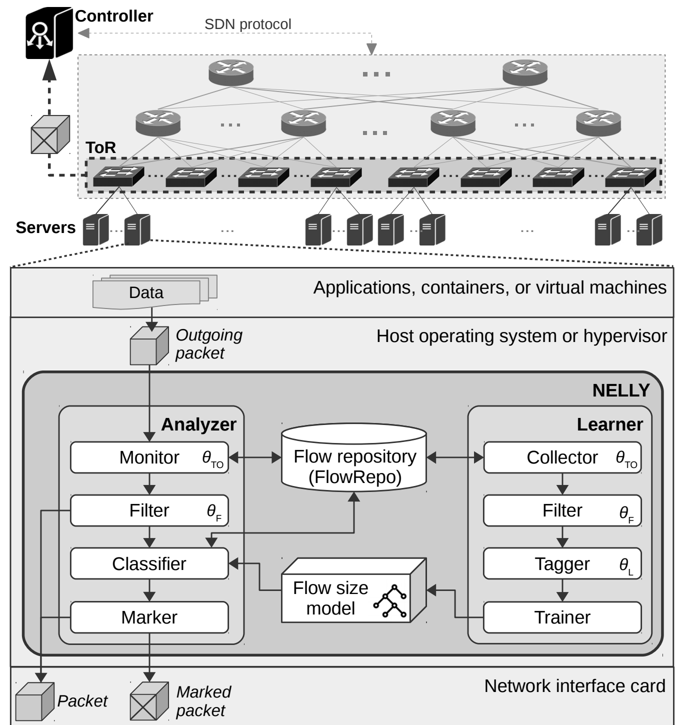



Figure 1. NELLY Architecture

NELLY is conceived for recognizing and handling elephant flows in real SDN implementations. NELLY can run on any host operating system or hypervisor. In the control plane, any OpenFlow-compliant controller (e.g., OpenDaylight) can be used since NELLY operates at the server-side. In the data plane, OpenFlow-compliant switches (e.g., Open vSwitch) can be employed since NELLY requires only that the Top-of-Rack (ToR) switches include a pre-configured routing rule to forward elephant flows to the controller.

### A. Analyzer

As illustrated in Fig. 1, the Analyzer consists of four modules: *Monitor*, *Filter*, *Classifier*, and *Marker*. The process of each module is detailed in Algorithm 1. As shown in lines 1–2, the Monitor keeps track of flows by extracting the header, size, and timestamp of each outgoing packet. A flow consists of subsequent packets sharing the same value for certain header fields, and separated by a time-space shorter than a threshold timeout ($\theta_{TO}$). NELLY enables a flexible configuration of these flow parameters, namely, flow header fields and $\theta_{TO}$. For example, the flow header fields can be set as the well-known 5-tuple: source IP, source port, destination IP, destination port, and IP protocol. These flow header fields can also include MAC addresses and VLAN ID. On the other hand, the configuration of $\theta_{TO}$ is discussed in Section II-B.

The Analyzer manages a flow record in the Flow Repository (FlowRepo) for each observed flow. As illustrated in Fig. 2, the flow record includes the flow identifier (FlowID), start time, last-seen time, packet header (e.g., 5-tuple), flow size, the size and Inter-Arrival Time (IAT) of the first $N$ packets, as well as the identified class (i.e., mice or elephants). Note that the IAT

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



**Algorithm 1:** Analyzer

```
input : outgoing packet p with header h_p, size s_p, and timestamp t_p,
        and flow size classification model m
output: either packet p or packet marked p*
data  : flow timeout threshold θ_TO, filtering flow size threshold θ_F,
        and number of first packets N
1  begin on receiving p
       // Monitor
2      get h_p, s_p, and t_p from p;
3      fid ← compute FlowID using the flow header fields from h_p;
4      if fid ∉ FlowRepo then
5          f ← call CREATE_FLOW(fid, h_p, s_p, t_p)
6      else
7          F ← fetch the last flow f ∈ FlowRepo such that f.id = fid;
8          if (currentTime − f.lastSeenTime) > θ_TO then
9              f ← call CREATE_FLOW(fid, h_p, s_p, t_p)
10         else
11             f ← call UPDATE_FLOW(f, s_p, t_p)
12         end
13     end
       // Filter
14     if f.size < θ_F then return p;
       // Classifier
15     if ∄ f.class then
16         f.class ← m.CLASSIFY(f);
17         update f → FlowRepo;
18     end
       // Marker
19     if f.class = "Elephant" then
20         p* ← mark p;
21         return p*;
22     end
23     return p;
24 end
25 function CREATE_FLOW(fid, h_p, s_p, t_p):
26     f ← initialize a new flow with FlowID fid;
27     f.headerFields[] ← array of flow header fields from h_p;
28     f.startTime ← f.lastSeenTime ← t_p;
29     f.size ← f.sizePackets[0] ← s_p;
30     create f → FlowRepo;
31     return f
32 end
33 function UPDATE_FLOW(f, s_p, t_p):
34     n ← current number of packets of f;
35     if n ≤ N then
36         f.sizePackets[n] ← s_p;
37         f.iatPackets[n] ← t_p − −f.lastSeenTime;
38     end
39     f.size ← f.size + s_p;
40     f.lastSeenTime ← t_p;
41     update f → FlowRepo;
42     return f
43 end
```

of the first packet is not included because it does not provide distinctive flow information (i.e., the IAT is always zero for the first packet of every flow).

As depicted in lines 3–13 in Algorithm 1, the Monitor then generates a FlowID from the flow header fields of each packet and checks to see if it exists in the FlowRepo. If this FlowID is missing (e.g., for packets 1 and 3 in Fig. 2), or if the time since the last update of an existing record with this FlowID is longer than $\theta_{TO}$ (e.g., for packet 4), the Monitor creates a new record in the FlowRepo (Algorithm 1, lines 25–32). Otherwise, the Monitor fetches and updates the flow record (Algorithm 1, lines 33–43) using the FlowID stored in the FlowRepo (e.g., for packets 2 and 5 through 10 in Fig. 2). When multiple flow records sharing the same FlowID exist in the FlowRepo, the Monitor always works with the most recent one (e.g., for packets 5 to 10).

Using the updated flow record, the Filter (Algorithm 1, line 14) avoids the introduction of a delay in the classification of a large number of mouse flows (usually latency-sensitive [14], [15]) by sending the packets of flows with a size below a certain threshold ($\theta_F$) directly to the SDDCN without further processing (e.g., for packets 1 to 9 in Fig. 2). The Filter also ensures that the Classifier receives all the required online features for making the classification. The online features refer to flow data extracted from the first $N$ packets of a flow. The Filter then guarantees the size and IAT of the first $N$ packets of a flow since the maximum value of $N$ depends on $\theta_F$. For example, $\theta_F = 10$ kB would require an $N \leq 7$ over Ethernet, otherwise, data from some packets would be missed. Consequently, the Classifier operates once the Monitor has processed packets that increment the size of flows over $\theta_F$ (e.g., for packet 10).

The Classifier (Algorithm 1, lines 15–18) applies the flow size classification model to the online features to identify flows as either mice or elephants. This model results from an incremental learning algorithm, which maps the online features to the corresponding class of flows used as training data. After applying the flow size classification model, the Classifier stores the identified class in the FlowRepo for each flow record with flow size greater than $\theta_F$ (e.g., elephant for flow of packet 10 in Fig. 2). Therefore, when processing a packet of a previously identified flow, the Classifier checks the fetched class from the FlowRepo to avoid any delay from the classification. The Classifier then reports to the Marker the class of the flow for each packet. We discuss in Section II-B how the Learner collects the training data for building and updating the flow size classification model.

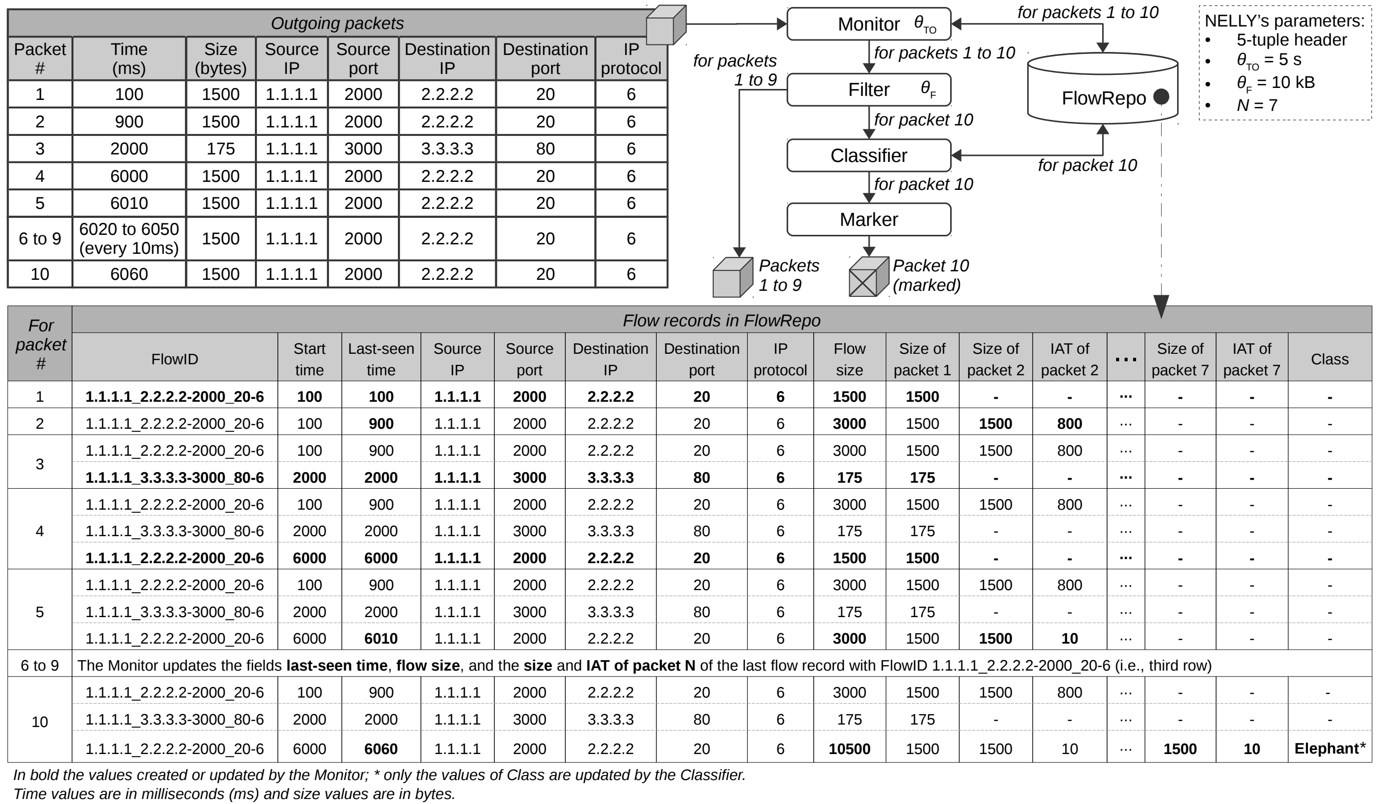
The Marker (Algorithm 1, lines 19–23) forwards the packets of flows classified as mice without changes but marks those classified as elephants (e.g., packet 10 in Fig. 2). To mark a packet, the Marker sets a predefined value in a code point header field supported by SDN switches. For example, OpenFlow switches support matching in two code point header fields. The first of these is the 6-bit Differentiated Services Code Point (DSCP) field of the IPv4 header. This DSCP reserves a code point space for experimental and local usage (i.e., $****11$, where $*$ is 0 or 1). The second is the 3-bit 802.1Q Priority Code Point (PCP) field of the Ethernet header. In practice, NELLY can rely on either one of these fields, since it is improbable that a data center use both DSCP and PCP simultaneously [15].

The Marker can be extended by enabling a flexible configuration of the number of subsequent packets in an elephant flow to be marked ($M$), thus enabling a trade-off between reliability and latency. For instance, as $M$ increases, the lesser the probability that the controller will miss elephant flows due to losses of marked packets in the SDDCN. However, a higher $M$ introduces a delay in the Marker for a higher number of packets of elephant flows. Once the controller has installed a higher priority routing rule for handling a specific elephant flow across the SDDCN, the subsequent marked packets of this flow are not forwarded to the controller.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291






| Outgoing packets | | | | | | | |
|---|---|---|---|---|---|---|---|
| Packet # | Time (ms) | Size (bytes) | Source IP | Source port | Destination IP | Destination port | IP protocol |
| 1 | 100 | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |
| 2 | 900 | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |
| 3 | 2000 | 175 | 1.1.1.1 | 3000 | 3.3.3.3 | 80 | 6 |
| 4 | 6000 | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |
| 5 | 6010 | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |
| 6 to 9 | 6020 to 6050 (every 10ms) | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |
| 10 | 6060 | 1500 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 |

| For packet # | Flow records in FlowRepo | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FlowID | Start time | Last-seen time | Source IP | Source port | Destination IP | Destination port | IP protocol | Flow size | Size of packet 1 | Size of packet 2 | IAT of packet 2 | ... | Size of packet 7 | IAT of packet 7 | Class |
| 1 | **1.1.1.1_2.2.2.2-2000_20-6** | **100** | **100** | **1.1.1.1** | **2000** | **2.2.2.2** | **20** | **6** | **1500** | **1500** | - | - | ··· | - | - | - |
| 2 | 1.1.1.1_2.2.2.2-2000_20-6 | 100 | **900** | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | **3000** | 1500 | **1500** | **800** | ··· | - | - | - |
| 3 | 1.1.1.1_2.2.2.2-2000_20-6 | 100 | 900 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | 3000 | 1500 | 1500 | 800 | ··· | - | - | - |
| | **1.1.1.1_3.3.3.3-3000_80-6** | **2000** | **2000** | **1.1.1.1** | **3000** | **3.3.3.3** | **80** | **6** | **175** | **175** | - | - | ··· | - | - | - |
| 4 | 1.1.1.1_2.2.2.2-2000_20-6 | 100 | 900 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | 3000 | 1500 | 1500 | 800 | ··· | - | - | - |
| | 1.1.1.1_3.3.3.3-3000_80-6 | 2000 | 2000 | 1.1.1.1 | 3000 | 3.3.3.3 | 80 | 6 | 175 | 175 | - | - | ··· | - | - | - |
| | **1.1.1.1_2.2.2.2-2000_20-6** | **6000** | **6000** | **1.1.1.1** | **2000** | **2.2.2.2** | **20** | **6** | **1500** | **1500** | - | - | ··· | - | - | - |
| 5 | 1.1.1.1_2.2.2.2-2000_20-6 | 100 | 900 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | 3000 | 1500 | 1500 | 800 | ··· | - | - | - |
| | 1.1.1.1_3.3.3.3-3000_80-6 | 2000 | 2000 | 1.1.1.1 | 3000 | 3.3.3.3 | 80 | 6 | 175 | 175 | - | - | ··· | - | - | - |
| | 1.1.1.1_2.2.2.2-2000_20-6 | 6000 | **6010** | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | **3000** | 1500 | **1500** | **10** | ··· | - | - | - |
| 6 to 9 | The Monitor updates the fields **last-seen time**, **flow size**, and the **size** and **IAT of packet N** of the last flow record with FlowID 1.1.1.1_2.2.2.2-2000_20-6 (i.e., third row) | | | | | | | | | | | | | | | |
| 10 | 1.1.1.1_2.2.2.2-2000_20-6 | 100 | 900 | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | 3000 | 1500 | 1500 | 800 | ··· | - | - | - |
| | 1.1.1.1_3.3.3.3-3000_80-6 | 2000 | 2000 | 1.1.1.1 | 3000 | 3.3.3.3 | 80 | 6 | 175 | 175 | - | - | ··· | - | - | - |
| | 1.1.1.1_2.2.2.2-2000_20-6 | 6000 | **6060** | 1.1.1.1 | 2000 | 2.2.2.2 | 20 | 6 | **10500** | 1500 | 1500 | 10 | ··· | **1500** | **10** | **Elephant*** |

*In bold the values created or updated by the Monitor; * only the values of Class are updated by the Classifier.*
*Time values are in milliseconds (ms) and size values are in bytes.*

Figure 2. Example of how flow records are created and updated in the FlowRepo

## B. Learner

As depicted in Fig. 1, the Learner consists of four modules: *Collector*, *Filter*, *Tagger*, and *Trainer*. The process of each module is detailed in Algorithm 2. As shown in lines 1–4, the Collector fetches terminated flows from the FlowRepo at every interval $T$. A flow is considered terminated if it remains idle for longer than $\theta_{TO}$. Therefore, the Collector recognizes terminated flows by checking that a time longer than $\theta_{TO}$ has passed since the last-seen time of the FlowID records in the FlowRepo. Note that the Collector relies on the FlowID records updated by the Monitor for the recognition of the terminated flows, so their actual size can be obtained.

The Collector avoids increasing memory consumption in NELLY by removing terminated flows from the FlowRepo (Algorithm 2, line 5). The actual size of terminated flows can also be further used to provide fixed-memory probability distributions that support autonomous configuration of flow size thresholds. Memory requirements in the FlowRepo thus depend on both $T$ and $\theta_{TO}$. $T$ provides a trade-off between memory and processing. As $T$ decreases, the Collector removes the terminated flows from the FlowRepo more quickly, consuming less memory, but leading to more processing. In turn, $\theta_{TO}$ directly affects the number of FlowID records stored in memory. As $\theta_{TO}$ increases, the FlowRepo retains FlowID records for a longer time. $\theta_{TO}$ is related to the *inactive timeout* configuration of flow rules in SDN-enabled switches, which provides a trade-off between flow table occupancy and miss-rate (i.e., when the packet IAT is greater than the timeout) [22].

The Filter of the Learner (Algorithm 2, line 6) receives the terminated flows from the Collector and reports to the Tagger only those with size greater than $\theta_F$. The terminated flows are then used by the Trainer to build the flow size classification model. Since the Classifier operates only with flows of a size greater than $\theta_F$, the Filter of the Learner prevents the introduction of noise to the model.

The Tagger (Algorithm 2, lines 7–8) compares the actual size of the filtered flows to a labeling threshold ($\theta_L$) so that they can be tagged as either mice or elephants. $\theta_L$ will

**Algorithm 2:** Learner

**input :** flow size classification model $m$
**output:** either actual $m$ or updated $m$
**data :** learning time interval $T$, flow timeout threshold $\theta_{TO}$, filtering flow size threshold $\theta_F$, and labeling flow size threshold $\theta_L$

1 **begin** every $T$
  // Collector
2   $F \leftarrow$ **fetch** flows $f \in$ *FlowRepo*;
3   **for** $f \in F$ **do**
4     **if** *currentTime* $-$ $f$.*lastSeenTime* $> \theta_{TO}$ **then**
5       **delete** $f \rightarrow$ *FlowRepo*;
        // Filter
6       **if** $f$.*size* $\geq \theta_F$ **then**
          // Tagger
7         **if** $f$.*size* $\geq \theta_L$ **then** $f$.*class* $\leftarrow$ "Elephant" ;
8         **else** $f$.*class* $\leftarrow$ "Mouse" ;
          // Trainer
9         $m \leftarrow m$.UPDATE($f$.*headerFields*[], $f$.*sizePackets*[], $f$.*iatPackets*[], $f$.*class*);
10       **end**
11     **end**
12   **end**
13   **return** $m$;
14 **end**

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291


vary (e.g., 100 kB or 1 MB) as a function of the traffic characteristics and performance requirements of SDDCNs. Labeled flows provide the Trainer (Algorithm 2, line 9) with the *ground truth* for building a supervised learning model for flow size classification [17]. This classification model maps online features (i.e., packet header, size, and IAT of the first $N$ packets) onto the corresponding class (i.e., mice or elephants). Recall that the Classifier relies on the flow size classification model to identify elephant flows.

Since flows represent continuous and dynamic data streams, the Trainer uses an incremental learning algorithm (e.g., Hoeffding tree and online ensembles) for building the flow size classification model. Incremental learning enables updating the flow size classification model as the Trainer receives labeled flows over time, rather than retraining from the beginning [17], [21]. Therefore, NELLY adapts to varying traffic characteristics and performs continuous learning with limited memory resources. There is no need for the Trainer to maintain labeled flows in memory. This is an important characteristic of NELLY, since it helps to reduce the consumption of resources in all the servers of the SDDCN.

## III. Evaluation

This section presents the evaluation of NELLY in relation to accuracy and classification time by using real packet traces and incremental learning algorithms. A generic approach for designing ML-based solutions in networking [17] is used to describe and conduct the evaluation of NELLY.

### A. Datasets

Two real packet traces, UNI1 and UNI2, captured in university data centers [23], were employed to evaluate NELLY (Table I summarizes their characteristics). These two traces are shorter than three hours long, but their mice and elephants distributions are similar to those found in non-public traces collected at different periods along the day [8], [9]. On the other hand, to the best of our knowledge, neither traces nor datasets of IPv6 traffic in DCNs are publicly available. In line with that, NELLY was evaluated using IPv4 traffic only which represents over 99% of the packets in UNI1 and UNI2.

Only the following parameters needed to be defined to generate the datasets: the flow header fields, $\theta_{TO}$, and $N$. Firstly, the 5-tuple header (i.e., source IP, destination IP, source port, destination port, and IP protocol) as the flow header fields since it sufficiently characterizes IPv4 flows; hereinafter, they are referred just as flows. Secondly, $\theta_{TO} = 5$ s was established on the basis of the break-even point analysis between the flow table occupancy and the miss-rate in OpenFlow switches for DCNs considered by [22]. Then, since the maximum value of $N$ depends on $\theta_F$, $N = 7$ was set as the maximum for $\theta_F = 10$ kB. As shown in Table I, the selected $\theta_F$ encompasses all the potential elephants (i.e., flows carrying more than 95% of the traffic) and avoids the introduction of the classification delay to mice (for more than 93% of the flows). Using these parameters, the UNI1 and UNI2 data traces were processed to generate the corresponding flow size datasets, each containing somewhat more than a million flows (see Table I). Since NELLY only classifies flows greater than $\theta_F$, those smaller than $\theta_F = 10$ kB were removed from both datasets. Therefore, the UNI1 and UNI2 datasets consisted of approximately 70,000 and 60,000 flows, respectively.

The datasets [24] included the following flow information: start time, end time, 5-tuple header, size and IAT of the first 7 packets, as well as flow size. The start and end times enabled a more realistic evaluation (see Section III-C). The 5-tuple header and the size and IAT of the first 7 packets represented the online features for the flow size classification model. The flow size is compared to different $\theta_L$ (e.g., 50 kB, 100 kB, and 500 kB) to label the flows as mice or elephants (i.e., classes of interest). Unless otherwise stated, the datasets with $\theta_L =$ 100 kB were used. Labeled flows represented the ground truth for learning and validating the flow size classification model.

For feature engineering [17], various different data types were considered for the online features, particularly for the 5-tuple header. Certainly, the size and IAT of the first 7 packets (13 features, since the IAT of the first packet is not included) indicate a measurement, hence, numeric data, whereas the 5-tuple header contains two IP addresses in dotted-decimal notation (i.e., categorical data) and three numeric codes (i.e., nominal data). However, the huge set of possible categories for IP addresses (i.e., $2^{32}$) hinders a real implementation. To address this problem, the IP addresses were divided into four octets, resulting in a total of 11 nominal features for the 5-tuple header. To handle these 11 nominal features as numeric data, a *Numeric* (Num) header type was defined. These features were then transformed into binary digits (bits), generating 104 features for the 5-tuple header. Considering these binary features, two more header types were defined: *Binary-Numeric* (BinNum) to treat binary features as numeric data (i.e., a value between zero and one) and *Binary-Nominal* (BinNom) to handle binary features as nominal data (i.e., zero or one). Unless otherwise stated, the datasets with BinNom-header were used.

### B. Accuracy metrics

Metrics derived from the confusion matrix were used, including the True Positive Rate (TPR) and the False Positive Rate (FPR), thus avoiding the over-optimism of the conventional accuracy metric caused by an imbalance of classes [17]. In the datasets, the imbalance between mice and elephants depends on $\theta_L$. For example, assuming $\theta_L = 100$ kB, only 12% of flows above 10 kB in the UNI1 dataset represent the elephant class (see Table I). The imbalance grows as $\theta_L$ increases.

Recall that flows classified as elephants are forwarded to the controller for further processing, thus introducing transmission and processing delays. Therefore, NELLY aims at detecting as many elephants while negatively affecting as few latency-sensitive mice as possible. Considering elephants as the positive condition, the TPR describes the proportion of detected elephants whereas the FPR provides the ratio of negatively affected mice. Both TPR and FPR range between 0 and 1. Furthermore, the Matthews Correlation Coefficient (MCC) was used to analyze the balance between the TPR and the FPR.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



Table I
DETAILS OF PACKET TRACES AND IPV4 FLOWS OBTAINED USING THE 5-TUPLE HEADER AND $\theta_{TO} = 5$ S

| **Packet traces** [23] | | **UNI1** | | **UNI2** | |
|---|---|---|---|---|---|
| **Duration** | | 65 min | | 158 min | |
| **Packets** | | 19.85 M | | 100 M | |
| **IPv4 % of total traffic** | | 98.98% *(mostly TCP)* | | 99.9% *(mostly UDP)* | |
| **IPv4 flows** | | 1.02 M *(TCP and UDP)* | | 1.04 M *(mostly UDP)* | |
| **Details of IPv4 flows** | **Flow size** | **% of IPv4 flows** | **% of IPv4 traffic** | **% of IPv4 flows** | **% of IPv4 traffic** |
| | $\geq$ **10 kB** | 7.16% | 95.06% | 5.91% | 98.81% |
| | $\geq$ **100 kB** | 0.83% | 83.71% | 1.93% | 96.86% |
| | $\geq$ **500 kB** | 0.14% | 73.14% | 0.76% | 93.52% |
| | $\geq$ **1 MB** | 0.07% | 69.52% | 0.48% | 90.83% |
| | $\geq$ **5 MB** | 0.01% | 60.33% | 0.17% | 81.34% |

The MCC takes all values from the confusion matrix to provide a measure between 1 and -1. As the MCC gets closer to 1, the difference of the TPR over the FPR increases, leading to a more accurate classifier. An MCC between 0 and -1 means that TPR $\leq$ FPR, which would be less accurate than a random classifier. In our experiment, the MCC values were always greater than 0, hence, we use a range between 0 and 1 to plot TPR, FPR, and MCC in Figures 3 and 4.

The MCC metric is employed in the performance analysis because it is recommended for imbalanced datasets (like UNI1 and UNI2) [25]. The MCC score is only high when the classification algorithms are doing well in both the positive and negative elements (i.e., elephants and mice, respectively). The ROC curve has also proven to be useful for imbalanced datasets but it is more appropriate to analyze classification algorithms that output a real value [26]. Thus, we preferred the MCC because the output of the incremental learning classification algorithms employed in this paper is a single class value (either mouse or elephant) rather than real value.

### C. Experiment setup

Incremental learning algorithms are commonly evaluated using the interleaved test-then-train approach [27]. This approach refers to going through each flow to classify it first by working only with the online features and then use its actual class for training the flow size classification model. However, since flows start and end over time, some order of the flows must be established. Moreover, under real conditions, some flows start before a classified flow ends, whereas others end before a new flow starts. Therefore, the flows are classified at the start time and the model is trained at the end time, so the performance evaluation will be based on more realistic conditions.

The imbalance of classes in the UNI1 and UNI2 datasets was addressed by training the flow size classification model using *inverse weights*, as in [18], i.e., weights (between 0 and 1) inversely proportional to the ratio of training instances previously encountered by the model for each class. If the model is trained with a *single weight* (i.e., 1 by default in the Massive Online Analysis (MOA) tool [27]), it would tend to classify all flows as mice due to the imbalance of classes.

### D. Performance analysis

To determine the consideration for the best performance of NELLY, the UNI1 and UNI2 datasets were used with different header types (i.e., Num, BinNum, and BinNom), as well as 50 incremental learning classification algorithms available in MOA. The performance evaluation included the accuracy metrics (i.e., TPR, FPR, and MCC) and the classification time per flow ($T_C$). The algorithms were executed with their default settings (except for the training weights) and without previous model initialization.

For the sake of brevity, Table II presents ten algorithms, namely, Adaptive Hoeffding Option Tree (AHOT), Adaptive Random Forest (ARF), Hoeffding tree, k-Nearest Neighbors (kNN) with Probabilistic Adaptive Windowing (PAW), Naive Bayes (NB), Online Accuracy Updated Ensemble (OAUE), OzaBag, Oza and Russell's Bagging (OzaBag) and Boosting (OzaBoost), Rule classifier with NB (Rule-NB), and Stochastic Gradient Descent (SGD) for Support Vector Machines (SVM). These algorithms were selected on the basis of the best performance results between algorithms with a similar learning approach. Furthermore, Table II includes only the best results of each algorithm, taking into account both accuracy and classification time for a specific header type. The BinNom headers were found to enable the best performance of the majority of the algorithms for the UNI1 and UNI2 datasets. This was due to the fact that most algorithms achieved greater accuracy using the BinNom headers than the Num headers for a comparable classification time. The use of the BinNum headers is strongly discouraged; although similar or slightly better accuracy results were obtained, there was a significant increase in the classification time (up to 4x).

The accuracy results show that no single algorithm achieves the best values of the TPR and MCC for the UNI1 and UNI2 datasets. This is due to the fact that the flow size distribution and the features of the elephant and mouse flows were specific for each dataset. Therefore, the top five results were used to analyze the accuracy performance. Regarding the $T_C$, most algorithms introduced a classification delay per flow shorter than 17.5 $\mu$s, but this represents only a small percentage (7%) of the Round-Trip Time (RTT) in DCNs (i.e., 250 $\mu$s in the absence of queuing [28]).

Both Hoeffding tree and NB represent the state-of-the-art in incremental learning algorithms. Their simplicity and low computational cost enabled a very short delay ($T_C < 5$ $\mu$s) that accounts for only 2% of the RTT in DCNs. However, only the Hoeffding tree represents a valid alternative for the traffic similar to that of UNI1 because its TPR and MCC were among the top five results for the UNI1 dataset. The Hoeffding tree in MOA uses NB classifiers on the leaves

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



Table II
PERFORMANCE OF NELLY WITH DIFFERENT INCREMENTAL LEARNING ALGORITHMS FOR CLASSIFYING FLOWS AS MICE AND ELEPHANTS

| **Algorithms** | **UNI1** | | | | | **UNI2** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **TPR(%)** | **FPR(%)** | **MCC** | **$T_C(\mu s)$** | **Header type** | **TPR(%)** | **FPR(%)** | **MCC** | **$T_C(\mu s)$** | **Header type** |
| AHOT | **85.97** | 35.52 | **0.327** | **4.07** | BinNom | **60.16** | 28.58 | **0.304** | **10.17** | BinNom |
| ARF | **82.39** | 28.82 | **0.359** | **12.01** | BinNom | **68.65** | 21.33 | **0.460** | **17.39** | BinNom |
| Hoeffding tree | **86.79** | 36.38 | **0.326** | **3.18** | BinNom | 57.92 | 28.46 | 0.284 | **4.64** | BinNom |
| kNN-PAW | 25.30 | 2.99 | 0.311 | 473.1 | Num | 40.29 | 10.22 | 0.302 | 454.1 | Num |
| NB | 74.76 | 35.69 | 0.254 | **4.76** | BinNom | 49.74 | 23.18 | 0.267 | **4.82** | BinNom |
| OAUE | **86.79** | 33.63 | **0.347** | 25.58 | BinNom | **63.28** | 28.65 | **0.332** | 33.06 | BinNom |
| OzaBag | **87.78** | 37.11 | **0.327** | 23.98 | BinNom | **64.17** | 31.13 | **0.314** | 36.61 | BinNom |
| OzaBoost | 75.88 | 29.93 | 0.307 | **11.56** | BinNom | **64.62** | 32.22 | **0.307** | **16.82** | BinNom |
| Rule-NB | 74.44 | 33.77 | 0.267 | 18.47 | BinNom | 54.83 | 29.15 | 0.248 | 18.59 | BinNom |
| SGD-SVM | 16.76 | 10.21 | 0.067 | **0.81** | Num | 38.69 | 30.99 | 0.076 | **0.8** | Num |

*In bold the top five results of TPR and MCC, and the $T_C$ results shorter than 17.5 $\mu$s, for both UNI1 and UNI2*

(nodes), which improves the accuracy without compromising the computational cost.

The ARF, OAUE, OzaBag, and OzaBoost are ensemble-based algorithms that combine multiple Hoeffding trees (ten in our evaluation) for improving the accuracy at the expense of increasing the computational cost. The ARF and OzaBoost algorithms introduced a $T_C$ shorter than 7% of the RTT in DCNs. ARF provided the best MCC and a TPR among the top five for the UNI1 and UNI2 datasets. OzaBoost can be seen as an option for the traffic similar to that of UNI2 since it was in the top five accuracy results only for the UNI2 dataset. In contrast, although the OAUE and OzaBag algorithms also provided good accuracy results (particularly for the UNI1 dataset), they introduced a $T_C$ twice longer than the $T_C$ of ARF and OzaBoost. This long $T_C$ is because OAUE and OzaBag rely on ensemble methods (block-weighting and bagging, respectively) that demand more computation than those used by ARF and OzaBoost (random forests and boosting, respectively).

Similar to ARF, the AHOT algorithm figured in the top five accuracy results for both datasets. Moreover, AHOT only introduced a $T_C$ shorter than 2% (5 $\mu$s) and 4% (10 $\mu$s) of the RTT in DCNs for the UNI1 and UNI2 datasets, respectively. AHOT is capable of improving the accuracy of the Hoeffding tree algorithm without demanding too much computation by providing an intermediate solution between a single Hoeffding tree and an ensemble of Hoeffding trees. AHOT uses additional option paths (five maximum in our evaluation) to build a single structure that efficiently represents multiple Hoeffding trees.

The implementations in MOA of the Rule-NB, SGD-SVM, and kNN-PAW algorithms are strongly discouraged. The Rule-NB presented accuracy results outside the top five for both datasets and a $T_C$ slightly longer than 7% of the RTT in DCNs. This is because rule-based algorithms focus on building more interpretable models than does the Hoeffding tree algorithm, which increases the computational cost but not necessarily improves the accuracy. The SGD-SVM algorithm introduced the shortest classification delay ($T_C < 1$ $\mu$s) but it produced the worst values in the TPR and MCC metrics. The reason for these values is that MOA implements a very simple SGD-SVM algorithm that uses a linear kernel which is not sufficient to model different patterns in flows of packets. The kNN-PAW provided the second-worst TPR for both datasets and a very long classification delay ($T_C > 450$ $\mu$s), which increased up to 3,000 $\mu$s with the BinNum and BinNom headers (i.e., 12x the RTT in DCNs). This long $T_C$ value is a consequence of the computation of a distance metric by the algorithms based on kNN every time the classification is performed.

In conclusion, NELLY achieves the best performance by using the BinNom headers along with the following incremental learning algorithms:

- The ARF is good for any type of traffic and if the RTT is flexible. It achieved the best MCC for the UNI1 and UNI2 datasets, and it was also the fifth- and second-best for the TPR while introduced a $T_C$ lesser than 7.5% of the RTT in DCNs.
- The AHOT is good for any type of traffic and a strict RTT. The TPR and MCC ranked among the top five for both datasets while the $T_C$ was shorter than that of the ARF, especially for the UNI1 dataset.
- The Hoeffding tree is good for traffic similar to that of UNI1 and if the RTT is very strict. The TPR was the second-best and the MCC was the fifth (quite close to the AHOT) for the UNI1 dataset while introduced a very short $T_C$. When the RTT constraint takes precedence over the accuracy, this would be a good option for traffic similar to that of UNI2 because a very short $T_C$ was maintained while provided the sixth-best TPR and MCC for such traffic.

The accuracy of NELLY was also evaluated with the ARF and AHOT algorithms for different values of $\theta_L$ since this threshold may vary as a function of traffic and routing requirements. Both ARF and AHOT ranked among the top five in accuracy for both datasets with a $T_C$ shorter than 7.5% of the RTT in DCNs. As shown in Fig. 3, the MCC results of both algorithms were degraded as $\theta_L$ increased, especially for the UNI1 dataset. This is because the difference between the features of the elephants and mice becomes less significant as $\theta_L$ increases. In contrast, the TPR remained very similar as $\theta_L$ increased, except that the ARF suffered from a significant reduction in the TPR for the UNI1 dataset. Therefore, the AHOT was more robust to variations in $\theta_L$ for traffic similar to that of UNI1, although the performance of both algorithms was similar for the UNI2 dataset. Based on this summary, NELLY with the AHOT algorithm enables a flexible configuration

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



of $\theta_L$ while providing great elephant flows detection in data centers regardless the type of traffic. For traffic similar to that of UNI2, both ARF and AHOT represent valid alternatives for the use of NELLY and the flexible configuration of $\theta_L$ is possible since they perform similarly in terms of elephants detection.

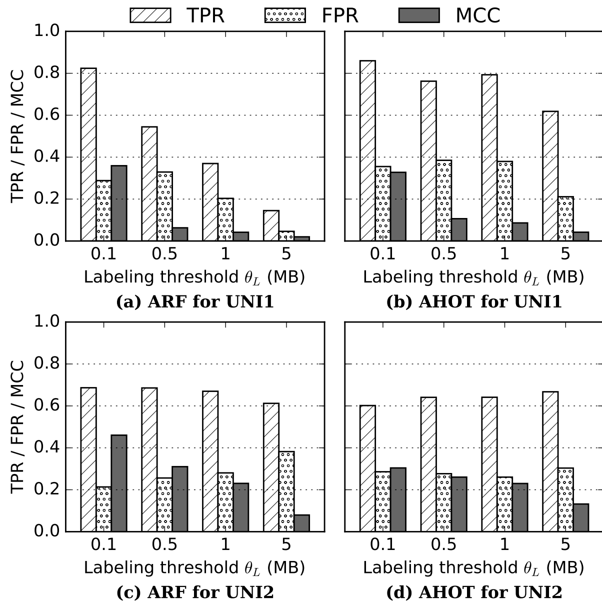



Figure 3. Accuracy of NELLY with the ARF (left) and AHOT (right) algorithms when varying the labeling threshold ($\theta_L$) for the UNI1 (top) and UNI2 (bottom) datasets.

Finally, the effect of the handling of different ranges of the inverse weights in the two classes on the accuracy of NELLY with the two algorithms (ARF and AHOT) was analyzed. The weights of the mice were maintained between 0 and 1, whereas the weights of the elephants ranged from 0 to $W_E$, where $W_E$ varied from 1 to 5. Fig. 4 shows that both the ARF and AHOT algorithms achieved a higher TPR for both datasets as $W_E$ increased (up to 94% and 98% of elephants detection, respectively). These results were expected since establishing greater weights for the elephant class makes the learning algorithms increment the influence of the features of the elephant flows in the classification model. Moreover, the trade-off between the TPR and FPR (i.e., MCC) remained quite similar for UNI1-type traffic whereas that of UNI2 was degraded as $W_E$ increased. This is due to the greater differences between the elephants and mice for the UNI1 dataset than for UNI2 when $\theta_L = 100$ kB. Therefore, as $W_E$ increased for the UNI2 dataset, the increment of mouse flows wrongly classified as elephants (i.e., FPR) was greater than that of elephant flows correctly classified (i.e., TPR). In conclusion, NELLY supports a flexible configuration of inverse weights for meeting different accuracy requirements.

## IV. Comparative Analysis

NELLY was compared with the Online Flow Size Prediction (OFSP) [18], the Efficient Sampling and Classification Approach (ESCA) [19], FlowSeer [20], and Mahout [15]. OFSP, ESCA, and FlowSeer incorporate ML at the controller-side of SDDCNs for proactively detecting elephant flows, whereas Mahout performs reactive detection at the server-side. The results reported by each work for the UNI1 dataset were used to compare them in relation to: learning approach, elephants detection, false elephants, table occupancy, control overhead, detection time, network modifications, and performance factors. The works involving Hedera [16] and DevoFlow [14] were not considered. These approaches perform reactive flow detection and their limitations hinder real implementation. Hedera causes large control traffic overhead and has poor scalability, whereas Devoflow requires custom-made switch hardware and imposes a heavy burden on switches.

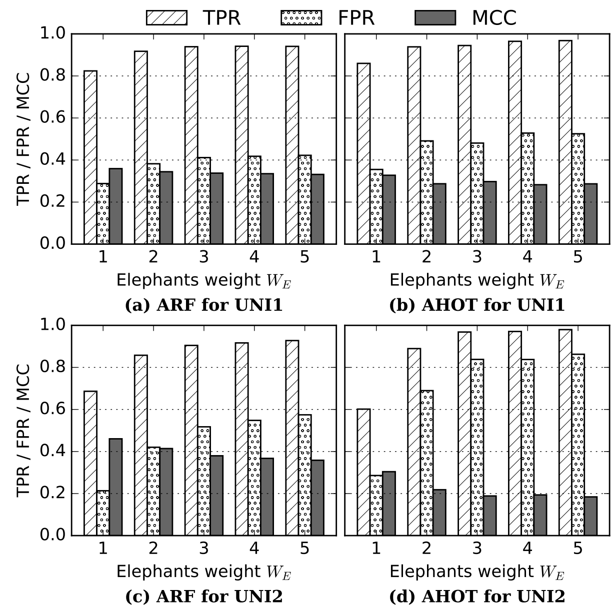



Figure 4. Accuracy of NELLY with the ARF (left) and AHOT (right) algorithms when varying the range for the inverse weights of elephant flows ($W_E$) for the UNI1 (top) and UNI2 (bottom) datasets.

**Learning approach.** ML algorithms used for detecting elephant flows can involve batch or incremental learning. Batch learning refers to the use of training models based on static datasets (i.e., all training data are simultaneously available). However, batch learning requires the storage of unprocessed data to cope with traffic variations in DCNs, so the models must repeatedly work from scratch. This is time-consuming and prone to outdated models. Conversely, incremental learning continuously adapts the ML models on the basis of streams of training data, enabling constantly updated models and reducing time and memory requirements [17], [21]. ESCA relies on batch learning whereas NELLY and OFSP rely on incremental learning for detecting elephant flows. FlowSeer is a mixed approach using batch learning for the identification of potential elephants and incremental learning for the classification of the potential ones. Mahout has no learning approach, since it performs reactive elephants detection.

**Elephants detection.** The main goal of flow detection methods is to identify elephant flows (i.e., TPR). NELLY, OFSP, and FlowSeer all proactively detected more than 95% of elephant flows, whereas ESCA detected a maximum of 88.3%.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291


Mahout provides perfect detection, although this is reactive.

**False elephants.** Mouse flows mistakenly identified as elephants (i.e., FPR) are needlessly forwarded to and processed by the controller. For achieving the highest elephants detection rate, FlowSeer informed the controller of 29% of mice as potential elephants, whereas OFSP and ESCA only reported around 2%. NELLY yielded an FPR of 40%, but this was computed using only 7% of the flows (i.e., $\theta_L \geq 10$ kB). NELLY thus forwards only 2.5% of mice to the controller. No mouse flow is reported to the controller by Mahout since detection is reactive.

**Table occupancy.** Controller-side flow detection methods install flow table entries in ToR switches for centrally collecting flow data. The smaller the number of flow table entries, the more efficient is the resource utilization. OFSP requires one entry per flow, thus constraining its scalability because of the limited memory in SDN switches. ESCA and FlowSeer install wildcard entries for sampling packets of flows. They reported 236 and 50 flow table entries, respectively, for achieving their highest detection rate in the UNI1 dataset. Conversely, NELLY and Mahout do not require flow table entries for collecting data since they operate at the server-side.

**Control overhead.** Flow detection methods require ToR switches to send control packets to the controller, either for the collection of flow data or for the reporting of detected elephant flows. The smaller the control overhead, the lower are the link utilization and the impact on the controller performance (since it has to process fewer control packets). The overhead of this control was computed by assuming no loss in the network and a control packet of 64 bytes. OFSP collects information from the first three packets of each flow, generating a control overhead of 402 kbps. FlowSeer collects information from the first five packets of sampled flows (i.e., 30% of the flow data) and potential elephants, yielding a control overhead of 288 kbps. ESCA reduces the control overhead to 215 kbps by using a sampling method that only reports information from the first packet. In contrast, NELLY and Mahout merely require that ToR switches send information of flows marked as elephants, greatly reducing the control message overhead to 4.4 kbps and 1.1 kbps, respectively.

**Detection time.** Timely detection of elephant flows enables the controller to make early decisions to improve routing. OFSP, ESCA, FlowSeer, and NELLY enable a short detection time by proactively detecting elephant flows. ESCA reported a detection time of 1.98 s for achieving the highest detection rate. OFSP and NELLY detect elephants in a shorter time since they rely on the first $N$ packets. On average, the detection time was 0.5 s for OFSP ($N = 3$) and 0.8 s for NELLY ($N = 7$). Further experimentation is needed to evaluate the detection time of FlowSeer. Nevertheless, the detection time of the latter would be slightly greater than for ESCA, since it is also based on sampling and considers the first five packets (ESCA considers only one packet). In contrast, Mahout relies on a reactive mechanism that detects elephant flows after their corresponding socket buffer in a server surpasses a certain threshold. Assuming a small threshold of 100 kB, the average detection time of Mahout is 3.8 s. However, unlike ML-based flow detection methods, the detection time of Mahout becomes longer as the threshold increases, which may cause hot-spots before the traffic carried by elephant flows reaches the threshold.

**Network modifications.** ESCA proposes a sampling method that depends on non-existing SDN specifications, hence, requiring custom-made switch hardware. In contrast, OFSP, FlowSeer, NELLY, and Mahout rely on OpenFlow [29], therefore enabling the use of commercial switches. Essentially, NELLY and Mahout require the installation of additional software in the servers, which need only to be done once with further configuration possible on the basis of a policy manager or autonomously. This installation can be carried out by using DevOps automation tools, such as Puppet and Chef, that support the distribution of software components to the operating systems of servers [30]. Moreover, virtualization platforms, such as VMWare and Xen, support software distribution to the servers as updates to the hypervisor without interrupting running virtual machines (either by live-migration or live-updating) [31].

**Performance factors.** Depending on the location of the flow detection method, different factors may affect its performance. Controller-side methods (i.e., OFSP, ESCA, and FlowSeer) rely on the resources available at the controller and ToR switches. The controller should be powerful enough for detecting all the elephants and processing the control packets sent by the ToR switches in the DCN. Similarly, the ToR switches should have enough memory for installing the required flow table entries. Moreover, the accuracy of the controller-side methods can be negatively affected if the ToR switches drop some of the first packets of the elephant flows. On the other hand, NELLY and Mahout operate at the server-side, so they depend on servers resources. As NELLY is based on ML, it requires more resources than does Mahout. Both server-side methods detect the elephants generated by each server (i.e., distributed operation). Note that servers should be able to monitor the first packets of the elephant flows for avoiding a decrease in accuracy.

## V. Conclusion and Future Directions

In this paper, we introduced NELLY to deal with the inaccuracy, large overhead, and poor scalability of current flow detection methods utilized in SDDCNs. NELLY is a novel flow detection method based on incremental learning that operates as a software component installed in every server of SDDCNs. An extensive evaluation demonstrated the accuracy and speed of NELLY, as well as its generation of low traffic overhead and adaptation to varying traffic characteristics. NELLY performs continuous learning and requires limited memory resources when used with the ARF and AHOT algorithms. The evaluation also corroborated the scalability of NELLY and the fact that no modifications in SDN standards are required.

As future work, we intend to implement NELLY as an in-kernel software component for evaluating its impact cost to server resources, including processing and memory consumption. Furthermore, we plan to evaluate NELLY in an emulated SDDCN by installing the software component into micro virtual machines connected to Open vSwitch instances.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TII.2019.2947291



Finally, although this paper has proven that incremental learning algorithms are efficient to detect elephant flows in DCNs, there are still research challenges to be addressed. First, there is no consistent and accepted method for defining the threshold value that discriminates between mice and elephants in DCNs. In this work, we evaluated different thresholds but did not specify how to select the appropriate threshold value for the traffic and routing requirements. Second, there is a need to create publicly available IPv6 dataset to allow the performance of ML-based elephant detection methods on such data set.

## References

[1] Z. Lv, H. Song, P. Basanta-Val, A. Steed, and M. Jo, "Next-generation big data analytics: State of the art, challenges, and future research topics," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1891–1899, Aug. 2017.

[2] M. Aazam, S. Zeadally, and K. A. Harras, "Deploying fog computing in industrial internet of things and industry 4.0," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4674–4682, Oct. 2018.

[3] D. A. Chekired, L. Khoukhi, and H. T. Mouftah, "Industrial iot data scheduling based on hierarchical fog computing: A key for enabling smart factory," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4590–4602, Oct. 2018.

[4] N. L. S. da Fonseca and R. Boutaba, *Cloud Services, Networking, and Management*, 1st ed. Hoboken, NJ, USA: John Wiley & Sons, 2015.

[5] S. K. Singh, T. Das, and A. Jukan, "A survey on internet multipath routing and provisioning," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2157–2175, Jul. 2015.

[6] M. Chiesa, G. Kindler, and M. Schapira, "Traffic engineering with equal-cost-multipath: An algorithmic perspective," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 779–792, Apr. 2017.

[7] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 123–137, Aug. 2015.

[8] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM IMC*, Melbourne, Australia, Nov. 2010, pp. 267–280.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *ACM IMC*, Chicago, IL, USA, Nov. 2009, pp. 202–208.

[10] M. H. ur Rehman, I. Yaqoob, K. Salah, M. Imran, P. P. Jayaraman, and C. Perera, "The role of big data analytics in industrial internet of things," *Future Generation Computer Systems*, vol. 99, pp. 247 – 259, Oct. 2019.

[11] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[12] F. Estrada-Solano, A. Ordonez, L. Z. Granville, and O. M. C. Rendon, "A framework for sdn integrated management based on a cim model and a vertical management plane," *Computer Communications*, vol. 102, pp. 150 – 164, Apr. 2017.

[13] F. Amezquita-Suarez, F. Estrada-Solano, N. L. S. da Fonseca, and O. M. C. Rendon, "An efficient mice flow routing algorithm for data centers based on software-defined networking," in *IEEE ICC*, Shanghai, China, May 2019, pp. 1–6.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[15] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM*, Shanghai, China, Apr. 2011, pp. 1629–1637.

[16] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *USENIX NSDI*, San Jose, CA, USA, Apr. 2010, pp. 19–19.

[17] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, Jun. 2018.

[18] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *IEEE ICNP*, Singapore, Nov. 2016, pp. 1–6.

[19] F. Tang, L. Li, L. Barolli, and C. Tang, "An efficient sampling and classification approach for flow detection in sdn-based big data centers," in *IEEE AINA*, Taipei, Taiwan, Mar. 2017, pp. 1106–1115.

[20] S. C. Chao, K. C. J. Lin, and M. S. Chen, "Flow classification for software-defined data centers using stream mining," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, Aug. 2016.

[21] S. Ayoubi, N. Limam, M. A. Salahuddin, N. Shahriar, R. Boutaba, F. Estrada-Solano, and O. M. Caicedo, "Machine learning for cognitive network management," *IEEE Commun. Mag.*, vol. 56, no. 1, pp. 158–165, Jan. 2018.

[22] A. Zarek, "Openflow timeouts demystified," Master's thesis, Department of Computer Science, University of Toronto, ON, Canada, 2012.

[23] T. Benson, "Data set for IMC 2010 data center measurement," University of Wisconsin-Madison, accessed Oct. 1, 2018. [Online]. Available: http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html

[24] F. Estrada-Solano, "NELLY datasets," GitHub, accessed May 29, 2019. [Online]. Available: https://github.com/festradasolano/nelly/tree/master/nelly-ml/analysis/datasets

[25] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using matthews correlation coefficient metric," *PLOS ONE*, vol. 12, no. 6, pp. 1–17, Jun. 2017.

[26] D. Chicco, "Ten quick tips for machine learning in computational biology," *BioData Mining*, vol. 10, no. 35, pp. 1–17, Dec. 2017.

[27] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: massive online analysis," *Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, Aug. 2010.

[28] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 63–74, Aug. 2010.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[30] M. Miglierina, "Application deployment and management in the cloud," in *SYNASC*, Timisoara, Romania, Sep. 2014, pp. 422–428.

[31] F. F. Brasser, M. Bucicoiu, and A.-R. Sadeghi, "Swap and play: Live updating hypervisors and its application to xen," in *ACM CCSW*, Scottsdale, AZ, USA, Nov. 2014, pp. 33–44.

**Felipe Estrada-Solano** (S'13) received the Bachelor degree in electronics and telecommunications engineering and the M.Sc. degree in telematics engineering from the University of Cauca, Popayan, Colombia, in 2016 and 2010, respectively. He is currently pursuing the Ph.D. degree in telematics engineering at the University of Cauca and the Ph.D. degree in computer science at the State University of Campinas, Campinas, Brazil. His topics of interest include network and service management, cloud computing, network virtualization, network softwarization, and machine learning.

**Oscar M. Caicedo** (S'11–M'15) received the Bachelor degree in electronics and telecommunications engineering and the M.Sc. degree in telematics engineering from the University of Cauca, Popayan, Colombia, in 2001 and 2006, respectively, and the Ph.D. degree in computer science from the Federal University of Rio Grande do Sul, Porto Alegre, Brazil, in 2015. He is currently a Full Professor with the Department of Telematics, University of Cauca, where he is a member of the Telematics Engineering Group. His research interests include network and service management, network virtualization, and software-defined networking.

**Nelson L. S. da Fonseca** (M'88–SM'01) received the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 1994. He is currently a Full Professor with the Institute of Computing, State University of Campinas, Campinas, Brazil. He has published 400+ papers and supervised over 70+ graduate students. He is currently the Vice President of Technical and Educational Activities of the IEEE Communications Society (ComSoc). He served as the ComSoc Vice President of Publications, Vice President of Member Relations, Director of Conference Development, Director of Latin America Region, and Director of On-Line Services. He is the former Editor-in-Chief of IEEE Communications Surveys and Tutorials. He was a recipient of the 2012 IEEE ComSoc Joseph LoCicero Award for Exemplary Service to Publications, the Medal of the Chancellor of the University of Pisa in 2007, and the Elsevier Computer Network Journal Editor of Year 2001 Award.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.