Cardinality Estimation in a Virtualized Network Device Using Online Machine Learning

Reuven Cohen Yuval Nezri
Department of Computer Science
Technion
Haifa, Israel

***Abstract*—Cardinality estimation algorithms receive a stream of elements, with possible repetitions, and return the number of distinct elements in the stream. Such algorithms seek to minimize the required memory and CPU resource consumption at the price of inaccuracy in their output. In computer networks, cardinality estimation algorithms are mainly used for counting the number of distinct flows, and they are divided into two categories: sketching algorithms and sampling algorithms. Sketching algorithms require the processing of all packets, and they are therefore usually implemented by dedicated hardware. Sampling algorithms do not require processing of all packets, but they are known for their inaccuracy. In this work we identify one of the major drawbacks of sampling-based cardinality estimation algorithms: their inability to adapt to changes in flow size distribution. To address this problem, we propose a new sampling-based adaptive cardinality estimation framework, which uses online machine learning. We evaluate our framework using real traffic traces, and show significantly better accuracy compared to the best known sampling-based algorithms, for the same fraction of processed packets.**

I. Introduction

Network measurement plays an important role in the evolution of large scale networks. Traffic statistics, such as top-K [26] and heavy hitters [3], are important for crucial network management applications. Cardinality estimation is the problem of estimating the number of distinct elements in a data stream with repeated elements. In the field of network measurement, this problem is mainly related to counting the number of unique flows in an IP packet stream, where each flow consists of many packets that have a unique 5-tuple: source IP, destination IP, source port, destination port and Protocol. Finding the number of distinct flows is important for tracking the load imposed on a web server, detecting a potential Distributed Denial of Service (DDoS) attack, and discovering other network anomalies.

Cardinality estimation algorithms are roughly divided into two categories: sampling algorithms and sketching algorithms. Sampling algorithms process only a subset of the stream, and use statistical analysis for estimating the cardinality of the entire stream. Such algorithms are very efficient in terms of their processing time and memory consumption. However, they have a relatively high error rate [6], especially when the stream has a skewed (power-law) distribution [11].

Sketching algorithms (HLL [15], HLL++ [22]) are known to offer the best performance in terms of statistical accuracy and memory storage. However, their main disadvantage is that they must process all the packets in the data stream. For this reason they have to be implemented using dedicated hardware [25].

Software network devices are gaining popularity due to the rise of two new network architecture concepts: Software Defined Networking (SDN) and Network Function Virtualization (NFV). The transition from executing cardinality estimation algorithms by dedicated hardware to executing them by a general purpose CPU introduces new challenges. Even with a very small number of operations per packet, processing every stream packet by a general purpose CPU is inefficient and in many cases even impossible [1], [2]. Because it is usually impossible to process all the packets, sketching cannot be used. In such cases we must resort to sampling-based solutions.

Many sampling cardinality estimation algorithms were developed for database query optimization. Since database records can represent any type of data, these algorithms make no prior assumption on the underlying data distribution. However, when network traffic is considered, information about flow size distribution can be used to improve the estimation precision. For example, the traffic generated by a DNS server usually includes many short flows, while the traffic generated by a video server has a small number of much longer flows. If the two servers produce the same amount of data, we expect the cardinality of the former type of traffic (DNS) to be greater than that of the latter (video). A problem with this approach is that the distribution of traffic flows is likely to change; e.g., due to dynamic server provisioning in the case of virtual environments, or due to a Network/Transport layer attack. We believe that dynamic adaptation of the cardinality estimation algorithms to changes in the flow size distribution can introduce significant improvement in their accuracy, and propose to use machine learning (ML) to obtain such adaptivity.

In this paper we propose a novel sampling-based adaptive cardinality estimation framework, which incorporates online ML algorithms to adapt to changes in flow size distribution. We describe the framework and its parameters, and then discuss the selection of an online ML algorithm and its features. We evaluate our framework using real traffic traces and show its accuracy improvement over the best known sampling algorithms.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes the basic concepts of sampling-based cardinality estimation and online

learning. Section IV presents the proposed framework for adaptive sampling cardinality estimation. Section V presents an extensive evaluation of our framework and compares it with the best-known sampling-based cardinality estimation algorithms. Section VI concludes the paper.

II. Related Work

Many works address the cardinality estimation problem and propose efficient algorithms to solve it. A detailed survey of the problem and its various solutions is presented in [17].

In this section we first discuss sampling-based solutions and cardinality estimation algorithms whose estimation process is adaptive. Then, we discuss works that analyze the efficiency and accuracy of software-based sketches. Many of the works we discuss here refer to the more general problem of estimating the flow size distribution, namely, the number of flows that contain a specific number of packets. This problem can be easily reduced to cardinality estimation.

In [20], Hass et al. present a family of cardinality estimators based on the generalized jackknife technique. They present an "unsmoothed first-order jackknife estimator" (UJ1) and its "smoothed" version (SJ1). They also present an "unsmoothed second-order jackknife estimator" (UJ2) and its "smoothed" and "stabilized" versions (SJ2 and UJ2A).

In [6], Charikar et al. present a lower bound on the error of sampling-based estimators. This lower bound implies that one should process almost the entire stream in order to guarantee good estimation error over all possible inputs. Then, they present Guaranteed Error Estimator (GEE) and Adaptive Estimator (AE). GEE has an optimal error, which matches their proved bound. AE is a refinement of GEE, which adapts to the input distributions and obtains reduced error over low-skewed data.

Recent work by Deolalikar et al. [11] conducts an extensive comparison of 11 sampling-based cardinality estimators and tests their accuracy over power-law distributed data with different skew parameters. GEE, AE and UJ2A [20] are found to be the most accurate estimators over a variety of data distributions. We shall refer to these three algorithms in the evaluation of our framework (Section V).

In [13], Duffield et al. take advantage of the SYN flag in the TCP header to obtain additional information about flow size distribution. This method requires that most of the packets in the stream be TCP. This requirement is not trivial in real traffic, due to the increasing popularity of the new UDP-based transport layer protocols [21]. Our framework can also use the number of SYN packets to improve the estimation accuracy, but it does not rely only on protocol specific information.

In [7], a novel hybrid approach is presented. It combines Good-Turing frequency estimation [16], a sampling-based solution, and the HyperLogLog algorithm [15]. This solution benefits from the computational efficiency of sampling and from the memory efficiency of sketching, but its accuracy is bound to that of sampling-based algorithms.

Alipourfard et al. [1] show that in a software switch (OVS [28]), calculating stream statistics using a simple hash table achieves better throughput and latency. They claim that the main reason is the trade-off between memory and CPU consumption imposed by sketching algorithms. Maintaining a sketch usually requires high CPU consumption, mainly for hashing each packet multiple times. However, modern servers have significantly improved cache size and efficiency. Hence, they do not really benefit from the reduced memory usage. CPU consumption is observed in their later work [2] as the new bottleneck resource.

III. Preliminaries

As indicated in the previous sections, a software device should not use sketching. On the other hand, sampling-based algorithms are known to have relatively low accuracy. To address this trade-off, the framework proposed in this paper combines sampling with online machine learning (ML). We start with a short discussion of each of these concepts.

A. Estimating Stream Cardinality from a Sample

Before we discuss the details of estimating stream cardinality from a sample, we formally define the problem. Our notations are summarized in Table I. Let S be a stream of N packets belonging to a certain number of flows. Each flow typically contains many packets, and D denotes the number of unique flows in S. Consider a sample s of n packets, taken randomly from S, and let $q = \frac{n}{N}$ be the sampling rate. Let d represent the number of unique elements in s. Sampling based cardinality estimation is the process of providing an estimate $\hat{D}$ of D given only s and q.

A typical workflow of a sampling cardinality estimation algorithm over network traffic is as follows. The entire traffic stream S is divided into batches $[S_1, S_2, ..., S_i, ...]$, and each batch S_i is analyzed separately. A sample s_i of n_i packets is collected from S_i according to a specific sampling rate q, using some sampling method. While several sampling methods have been proposed in the past [14], in this paper we consider random sampling, where all packets are sampled with the same probability. After a sample of packets is collected from the batch, various statistical properties are calculated from it, and an estimation of the flow cardinality is produced.

A commonly used statistical property is f_i^j, namely, the number of flows that appear exactly j times in s_i. A special case is f_i^0, which represents the number of flows that appear

Symbol	Meaning
S_i	a batch of packets (part of the stream)
N_i	size of S_i (number of packets in S_i)
D_i	cardinality of S_i (number of flows in S_i)
s_i	a sample of packets taken from S_i
n_i	size of s_i (number of packets in s_i)
q_i	sampling rate $q_i = n_i/N_i$
d_i	cardinality of s_i (number of flows in s_i)
$\hat{D}_i$	estimation of D_i
f_i^j	number of flows that appear exactly j times in s_i (have exactly j packets in s_i)

TABLE I: Notations

in S_i but not in s_i. Since $D_i = d_i + f_i^0$, estimating D_i can be reduced to estimating f_i^0, i.e. the number of "hidden" flows.

B. Online Machine Learning

We now describe the standard ML procedures we use. To this end, we use the Scikit-Learn's abstraction of ML algorithms [27]. According to this abstraction, a supervised ML algorithm includes two basic operations:

- $fit(training\_examples, labels)$ – The fit operation receives a set of training examples and their matching labels, and returns a "trained ML model", which is capable of delivering predictions from unlabeled examples. Each example consists of a set of features, where every feature is a property of the data that contributes to the estimation. For example, to train a ML model to predict housing prices, the features could be the size of the house, the number of rooms and the neighborhood. The label is the desired output value, i.e., the actual price of each house. For clarification, we use the term "ML model" to describe an instance of an ML algorithm after it is provided with training data.
- $predict(example)$ – After an ML model is trained using the *fit()* operation, the predict operation is used to obtain an estimated value given only a set of features. In our housing prices example, *predict()* is provided with information about a specific house: its size, number of rooms and neighborhood, and returns its price estimation.

A typical ML algorithm usually receives the entire set of training examples in advance. However, processing the entire training dataset at once is not feasible when analyzing large data streams that cannot entirely fit into the memory, or when the training examples are only gradually available. To address this problem, online ML algorithms have been developed. Such algorithms use the following *partial_fit()* operation, instead of *fit()*:

- $partial\_fit(training\_example, label)$ – This operation is similar to *fit()*, except that it trains the model over a single or a small number of labeled training examples. While *fit()* is executed only once, at the beginning of the model's lifespan, *partial_fit()* can be invoked many times.

In contrast to *fit()*, *partial_fit()* allows the model to adapt over time. For example, in the context of housing prices, very often there are not enough examples of houses and their true market price. With *partial_fit()*, each sold house can be added to the model as a new example. If during the model's lifespan there is a trend to prefer houses in different neighborhoods, such a trend is likely to be reflected by the new training examples, and the model is likely to be more accurate compared to a model that receives all its training examples in advance.

IV. The Proposed Framework for Sampling-Based Adaptive Cardinality Estimation

A. Framework Description

Our sampling-based adaptive cardinality estimation framework uses a novel approach: combining sampling and online learning. Figure 1 describes the processing of a packet stream in detail. The stream is divided into batches of packets. Each batch is sampled, and selected features are extracted from the samples. The features are then passed to the $predict()$ operation of the online ML algorithm, which returns an estimation of the batch's cardinality (Figure 1(a)).

Once every $training\_rate$ batches, the entire batch is sent for training (Figure 1(b)). In the training phase, $partial\_fit()$ is provided with the batch's set of features and its cardinality. The exact cardinality of a batch can be calculated using a simple hash table. Since the focus of this work is on the framework itself, we intentionally do not specify which online ML algorithm is used. As shown in Section IV-C, different algorithms may be suitable for different framework use cases. Procedure 1 presents the pseudo code for processing a single batch as described above.

Procedure 1 Processing of a Batch

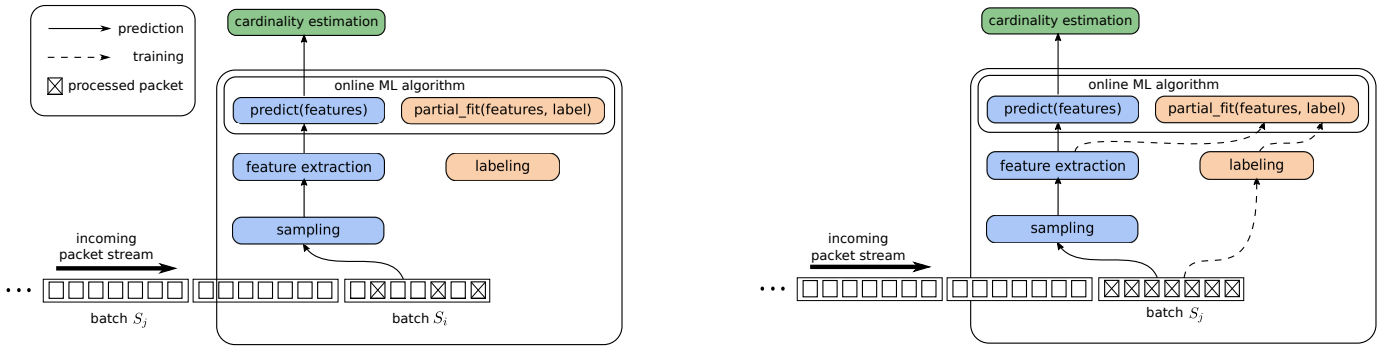
```
1:  batch_counter ← 0
2:  ml_model.init()
3:  function PROCESS_BATCH(batch)
4:      sample ← sample(batch, sampling_rate)
5:      features ← extract_features(sample)
6:      if is_training_batch(batch_counter, training_rate)
    then                                   ▷ executed concurrently
7:          label ← calculate_label(batch)
8:          ml_model.partial_fit(features, label)
9:      end if
10:     batch_counter ← batch_counter + 1
11:     return ml_model.predict(features)
12: end function
```

In steps 1 and 2, the batch counter and the online ML model are initiated. In steps 4 and 5, each batch is sampled and selected features are extracted from it. In step 6, batches are selected for training according to the chosen *training_rate* hyperparameter. In step 7, the true cardinality of training batches is calculated. In step 8, the training batch's feature set and true cardinality are input to the model's *partial_fit* operation. Finally, in step 11 an estimation of the batch's cardinality is returned. Since the training operations (steps 6–9) are resource and time consuming, they can be executed in the background without delaying the estimation.

B. Feature Selection

Choosing the right set of features is crucial to the accuracy of an ML algorithm. A good feature should be both informative and independent: an informative feature shows high correlation to the target value, while an independent feature shows low correlation to the other features.

The first "suspects" for such features are the f_i^j values, which are also used by statistical sampling-based algorithms. Recall that for a given batch S_i, f_i^j is the number of flows that are represented by exactly j packets in s_i. Figure 2 shows the relationship between f_i^1, f_i^2, f_i^3 and the cardinality of S_i (D_i). This graph is obtained using 300 batches, each of 100,000



(a) Prediction on batch S_i

(b) Prediction and training on batch S_j

Fig. 1: The proposed adaptive cardinality estimation framework

packets, taken from CAIDA-2016 traffic trace[1]. Batches are sampled with $q = 0.1$. We can see a strong linear correlation between f_i^1 and the cardinality. We can also learn that the correlation becomes weaker as j increases.

When we test the correlation between f_i^1 and the cardinality for other traces, we see that the linear correlation is usually conserved, but the slope and intercept vary, since they depend on the flow size distribution. For example, Figure 3(a) shows batch cardinality over time for the DARPA-DDoS trace [10]. This trace contains background traffic and a SYN flood DDoS attack on one target host. Each batch contains traffic collected during one second, and the average batch size is 11,228 packets. Batch no. 219 is removed from the trace since it shows abnormal behavior, which is irrelevant for the purpose of this example. We sample this trace with $q = 0.1$.

We can clearly divide the attack into 6 different stages, each beginning in one of the following time steps: $t = 0, 113, 163, 194, 223, 236$. Figure 3(b) shows the cardinality of the batch as a function of f^1, and Figure 4 describes the slope, intercept and Pearson correlation coefficient obtained by running a linear regression on each interval separately.

The Pearson correlation coefficient is a number between -1 and 1 that indicates the extent to which two variables are linearly related, where -1 indicates a perfect negative linear relation, 0 indicates no linear relation and 1 indicates a perfect positive linear relation. The Pearson correlation coefficient values in Figure 4 indicate a strong linear relation between f_i^1 and cardinality in most stages. The slope and intercept values indicate significant differences in the linear relation of different stages. We expect our online ML algorithm to identify these changes and to adjust its cardinality estimation accordingly.

While f_i^j, and in particular f_i^1, seem promising as features from which to learn about the cardinality, we can extract even more valuable information from a sample of IP packets. For example, sending a large amount of data usually involves long flows of relatively big packets. In the presence of such flows we expect the cardinality of the stream to decrease. Hence, we expect to find a negative correlation between the sample's average packet length and the stream's cardinality.

[1]Section V contains a detailed description of the CAIDA-2016 trace.

Moreover, when most of the traffic is TCP, we expect to find a correlation between the number of SYN packets in the sample and the stream cardinality, because each SYN packet usually represents a single TCP connection. Figure 5(a) shows the correlation between the average length of a packet in the sample (*avg_pkt_len*) and the batch cardinality, and Figure 5(b) shows the correlation between the number of SYN packets in the sample (*syn_count*) and the batch cardinality. Both figures are from the CAIDA-2016 trace with $q = 0.1$.

In our study we choose f_i^1, f_i^2, f_i^3, avg_pkt_len and syn_count as possible features, since they demonstrate good correlation to D and they can be extracted from the sample. However, any such property can be added as a feature to our framework.

C. Choosing an Online ML Algorithm

Since all the features we consider seem to present linear correlation with D, our first choice is a linear algorithm. Linear algorithms operate on assumption that the target value D is a linear combination of the features. Let $\vec{x}$ be the vector of features and $\vec{w}$ be a vector of linear coefficients, also known as the weights vector. In each estimation phase, the $predict(\vec{x})$ operation returns $\hat{D} = \vec{w} \cdot \vec{x}$. In each training phase, the $partial\_fit(\vec{x}, D)$ operation updates $\vec{w}$ according an update rule that aims to minimize a predefined notion of estimation error. This notion of prediction error is commonly called the loss function. The way the update rule and loss function are defined determines different properties of the online ML algorithm, such as:

- Aggressiveness – The intensity of updates in response to loss.
- Forgetting Rate – The effect of new training batches on the learning, compared to the effect of previous training batches.
- Outlier Sensitivity – How the algorithm responds to training batches that introduce extreme loss.

As shown in Figure 3(b), the linear relation between a single feature and flow cardinality may frequently change. These changes may be attributed, for example, to different stages of a DDoS attack. Thus, the chosen algorithm must be able

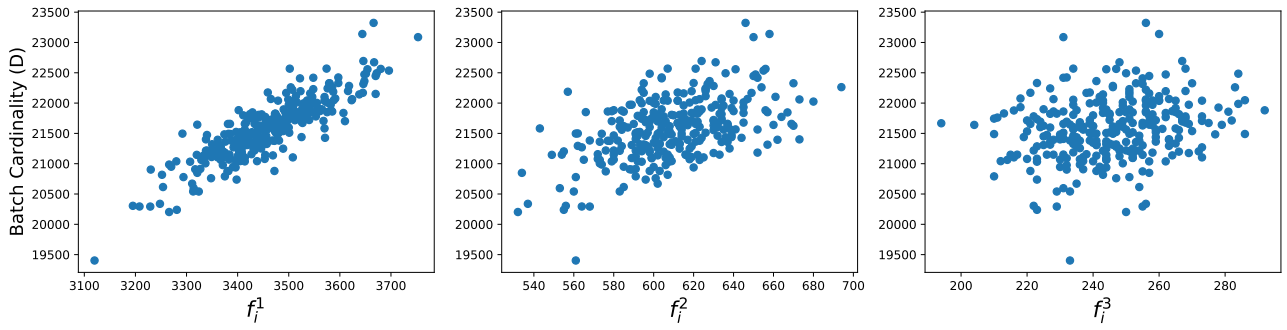
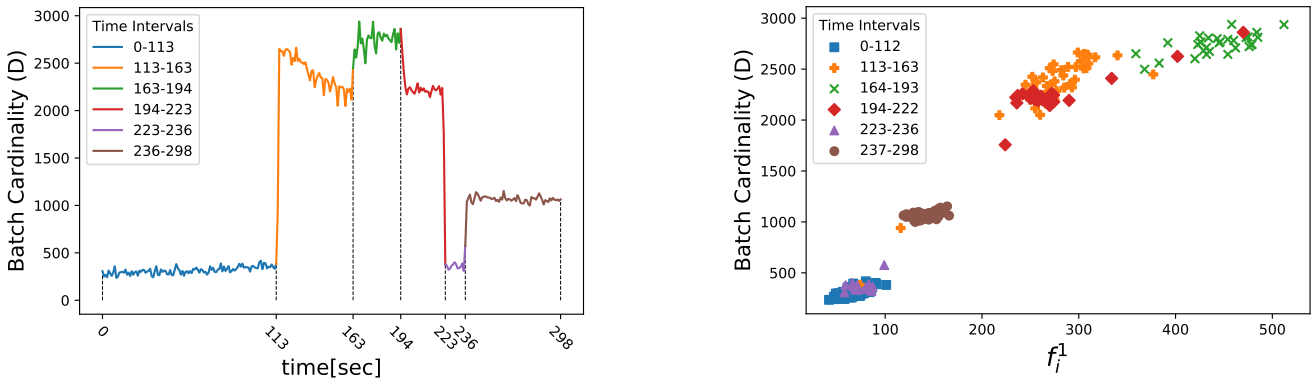


Fig. 2: The correlation between f_i^1, f_i^2, f_i^3 and flow cardinality ($q = 0.1$). We can see strong linear correlation between f_i^1 and the cardinality, and almost no correlation between f_i^3 and the cardinality



(a) Batch cardinality as a function of time

(b) Batch cardinality as a function of f_i^1

Fig. 3: The value of f_i^1 at different stages of the attack for DARPA-DDoS ($q = 0.1$)

Interval	Slope	Intercept	Pearson Coefficient
(0, 112)	2.49	144.17	0.68
(113, 163)	7.57	261.5	0.91
(164, 193)	1.84	1,928.54	0.67
(194, 222)	2.98	1,431.4	0.87
(223, 236)	2.99	149.88	0.54
(237, 298)	1.21	895.08	0.43

Fig. 4: Linear regression parameters of the different stages of the attack for DARPA-DDoS ($q = 0.1$)

to react rapidly to such changes. In Section V we compare 3 popular linear regression ML algorithms: Stochastic Gradient Descent (SGD) [23], Recursive Least Squares (RLS) [19] and Passive Aggressive (PA) [9].

We describe PA and RLS since we found them to be the best performing algorithms in our experiments. We also include SGD because it is the foundation for a wide family of online ML algorithms. We experimented with more advanced SGD based algorithms (RMSProp [29], ADAgrad [12] and ADAM [24]), but they are excluded from this work since they did not show significant improvement over the basic version (SGD).

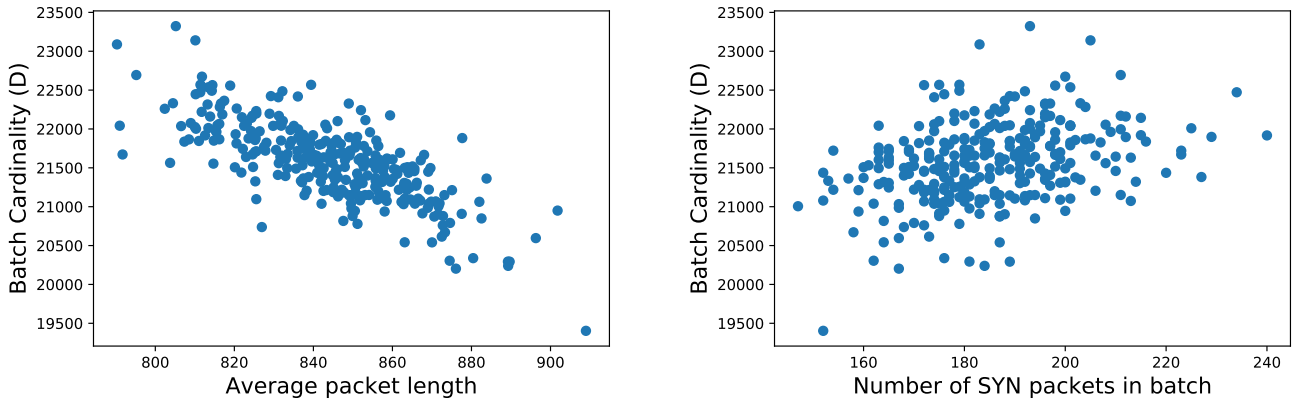
D. Rate Parameters

Obtaining timely and accurate estimation using low rate sampling is not trivial. Our framework has several rate parameters, which together have a crucial impact on the trade-off between accuracy and computational cost. Table II summarizes these parameters.

The traffic stream can be divided into batches using two different approaches. In the first approach, *batch_size* is fixed, and a prediction phase starts after *batch_size* packets are processed. This approach is used when the estimation is not time critical, and does not have to be provided during

Notation	Description
packet_rate	incoming packet rate
batch_size	size of each processed batch
sampling_rate	fraction of packets sampled from each batch
estimation_rate	number of cardinality estimations executed per second
training_rate	number of batches between subsequent training phases
sample_size	the size of a single sample
update_rate	number of training phases per second
effective_sampling_rate	the total fraction of packets being processed, including training phase

TABLE II: Main Parameters of the Proposed Framework



(a) Batch cardinality as a function of avg_pkt_len

(b) Batch cardinality as a function of syn_count

Fig. 5: The CAIDA-2016 trace with specific features ($q = 0.1$)

fixed time intervals. In the second approach, *estimation_rate* is fixed, and a prediction phase is invoked exactly every $1/$*estimation_rate* seconds. This approach is used when the time between consecutive estimations must be fixed.

When *batch_size* is fixed, *estimation_rate* is determined by the *packet_rate*/*batch_size* ratio. In this case, when *packet_rate* decreases, *estimation_rate* decreases as well. Eventually, if *packet_rate* is too low, we might end up with an *estimation_rate* that is not sufficiently high for our measurement application. For example, if the measurement application is DDoS detection, a measurement provided every 300 seconds might not be sufficiently frequent.

In addition, *estimation_rate* has an impact on the framework's *update_rate* since *update_rate* is determined by *estimation_rate* $\cdot$ *training_rate*. When *estimation_rate* is too low, the model may react too slowly to changes in the flow size distribution.

When *estimation_rate* is fixed, *batch_size* is determined by the *packet_rate*/*estimation_rate* ratio. In this case, when *packet_rate* decreases, *batch_size* and *sample_size* decrease as well, and we might end up with a statistically insufficient *sample_size*.

V. Performance Analysis

We evaluated our new framework using 4 different traffic traces. In this section we report our findings for all these traces. We compare the framework's accuracy to that of statistical sampling-based algorithms, and analyze the effect of the various online ML algorithms and framework parameters on the accuracy. We also discuss the trade-offs imposed by the various framework parameters.

A. Accuracy Metrics

We use the following metrics in our evaluation:

- Root Mean Squared Error (RMSE), defined as $\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(D_i - \hat{D}_i\right)^2}$. RMSE indicates the average prediction error in units of number of flows. Since the errors are squared before they are averaged, RMSE gives a higher weight to large errors, and penalizes high variance in the error distribution.
- Mean Absolute error (MAE), defined as $\frac{1}{n}\sum_{i=1}^{n}|D_i - \hat{D}_i|$. MAE also indicates the average prediction error in units of number of flows. As opposed to RMSE, it takes only the average error into account and is not affected by the error variance.
- Mean Absolute Percentage Error (MAPE), defined as $\frac{100}{n}\sum_{i=1}^{n}\left|\frac{D_i-\hat{D}_i}{D_i}\right|$. MAPE indicates the average error percentage. It is biased towards underestimations, since for such estimations the percentage error cannot exceed 100%, while for overestimations there is no upper limit. We use MAPE since it allows us to compare the accuracy over different traces and batch sizes even when batches have significant differences in their cardinality.
- Max Absolute Error (MAXAE), defined as $\max(|D_i - \hat{D}_i|)$. It indicates the maximum estimation error, and allows us to analyze the error in extreme cases.

B. Effective Sampling Rate

Both the proposed framework and statistical sampling-based algorithms use f_i^j values to produce an estimation. The computational resources required to obtain f_i^j from sampled packets are far greater than those required for calculating the estimation itself. Hence, to measure how much CPU is consumed by our framework, and to compare it to statistical sampling-based algorithms, we simply count the number of processed packets, while ignoring the CPU consumption for the execution of *predict()* and *partial_fit()*. In the same way, we ignore all calculations performed by the statistical sampling-based algorithms to which we compare our framework.

Statistical sampling-based algorithms process only the sampled packets, while our framework processes all sampled packets as well as complete training batches. Thus, in a

statistical sampling-based algorithm the expected number of processed packets in a batch is

$$mean\_batch\_size \cdot sampling\_rate,$$

while in our framework, this number is

$$mean\_batch\_size \cdot effective\_sampling\_rate$$

where,

$$\begin{aligned} effective\_sampling\_rate =& sampling\_rate + training\_rate \\ & - sampling\_rate \cdot training\_rate. \end{aligned}$$

Thus, the computational cost of statistical sampling-based algorithms is proportional to the *sampling_rate*, and the computational cost of our framework is proportional to the *effective_sampling_rate*. When we compare between the two methods in Section V, we ensure that the *sampling_rate* for the sampling-based algorithms is equal to the *effective_sampling_rate* of our framework.

C. Implementation Details

Our framework is designed to perform real-time cardinality estimation. However, in order to get reproducible and comparable results over all estimation methods, in our experiments data processing was performed in advance as follows[2]. First, all batches were sampled. Then, features and statistical properties were extracted from these samples, and the true cardinality of each batch was calculated. Then, these values were used to calculate an estimation and to train the online ML models. This guarantees that all estimators and online ML algorithms use the exact same samples and features for their estimations.

In steps 6–9 of Procedure 1, batch's true cardinality is computed concurrently with the *partial_fit()* operation. This concurrent execution prevents the training phase from delaying subsequent estimations. In our experiments this delay was insignificant, since the true cardinality of each batch was calculated in advance. Nonetheless, to express this behavior, during each training phase we first provided an estimation and only then trained the online ML model. Thus, it is assumed that the training phase ends before the subsequent batch estimation is requested.

Online ML algorithms rely on training over previously seen data examples, which are usually unavailable during the first stages of the model's lifespan. Hence, some kind of initialization process is required. This process of initializing the online ML algorithms is usually referred to as "bootstrapping". To bootstrap an online ML model we always used the first batch of packets as a training batch. Since each algorithm was implemented differently, a different bootstrapping process was used for each. For SGD we ran numerous *partial_fit()* iterations until the loss difference between two subsequent iterations became smaller than a predefined tolerance value, while for PA and RLS we found that a single *partial_fit()* operation is sufficient.

[2]The code used for our experiments can be found at https://github.com/yuvalnezri/CardEst.

D. The CAIDA-2016 Trace

The CAIDA-2016 [4] trace was collected from the Equinix-Chicago high-speed monitor over Internet backbone links. The part of this trace that we use contains 44,567,284 packets, collected during approximately 48 seconds.

Our analysis of the frequency distribution (f_i^j values) of the first batch shows a heavy-tailed behavior: 81% of the flows are small (contain less than 4 packets), while 51% of the packets belong to big flows (contain more than 20 packets). Other batches show similar distribution. Statistical sampling-based algorithms are known to demonstrate high error rates in heavy-tail distributed traffic [6], [20].

Figure 6 shows real vs. estimated cardinality, when the estimation is performed using 3 statistical sampling-based algorithms: GEE, AE and UJ2A. For these graphs, the *batch_size* is 100K and the *sampling_rate* is $q = 0.0199$. We can see that the prediction of all algorithms is far from the real cardinality.

Figure 7 shows real vs. estimated cardinality when the estimation is performed using our framework with 3 different online linear regression ML algorithms: Stochastic Gradient Descent (SGD), Recursive Least Squares (RLS) and Passive Aggressive (PA). In this graph we use *batch_size* $= 100K$, $q = 0.01$ and *training_rate* $= 0.01$. The feature set is only $\{f_i^1\}$. These parameters yield an *effective_sampling_rate* of 0.0199, which is identical to that used in Figure 6 for the sampling-based algorithms. In addition, we set the *learning_rate* of SGD to 10^{-6}, the forgetting factor of RLS to $\mu = 0.99$, the ϵ-insensitive loss function of PA to $\epsilon = 0.1$, and the aggressiveness parameter of PA-II update rule to $C = 1$. It can be clearly seen that the prediction of all the algorithms used in our framework is very close to the real cardinality.

Figure 8 summarizes the error rates of the experiments reported in Figure 6 and 7. Indeed, we see that the error rates for the statistical sampling-based algorithms are significantly higher than those of our new framework, regardless of the online ML algorithms being used. This is despite the fact that the same number of packets are processed in all cases.

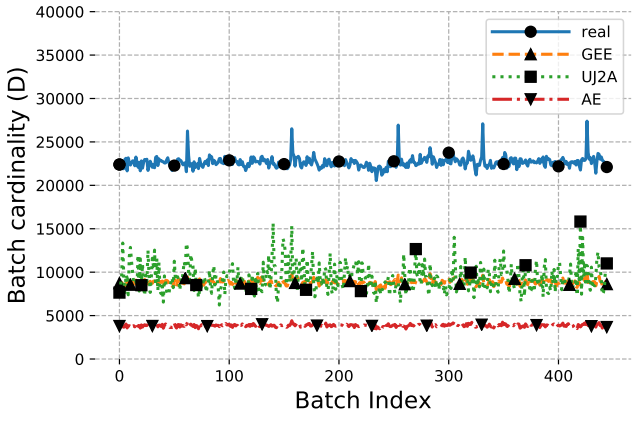


Fig. 6: Real vs. estimated batch cardinality of statistical sampling-based algorithms for the CAIDA-2016 trace

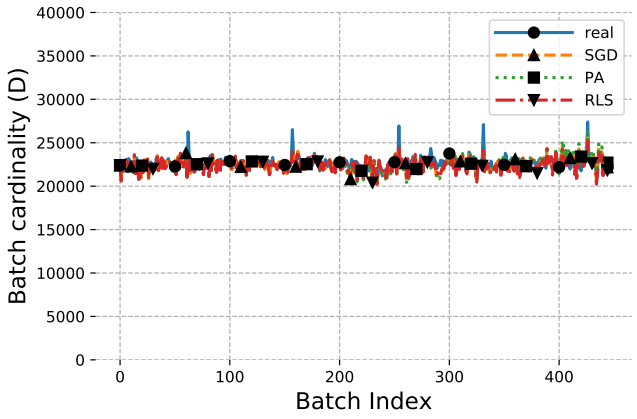


Fig. 7: Real vs. estimated batch cardinality of the proposed framework with different online ML algorithms for the CAIDA-2016 trace

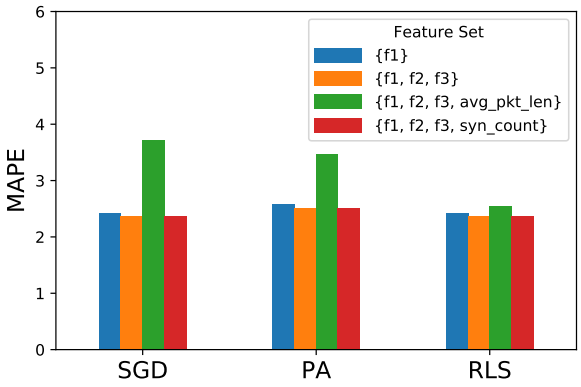


Fig. 9: The impact of extending the feature set on the accuracy of our framework for the CAIDA-2016 trace

In addition, we can see that when the flow size distribution hardly changes, the various online ML algorithms show similar accuracy. We also see that for this specific trace, AE is significantly less accurate than GEE and UJ2A. Overall, the proposed framework's MAPE is approximately 30 times better than the best performing statistical sampling-based algorithm.

Next, we examine the effect of adding more features to the online ML algorithm, as discussed in Section IV-B. Figure 9 depicts the MAPE of the three online ML algorithms with the following feature sets: $\{f_i^1\}$, $\{f_i^1, f_i^2, f_i^3\}$, $\{f_i^1, f_i^2, f_i^3, avg\_pkt\_len\}$ and $\{f_i^1, f_i^2, f_i^3, syn\_count\}$. We conclude from this graph that extending the features beyond $\{f_i^1\}$ does not yield significant improvement in the accuracy. Moreover, adding *avg_pkt_len* to the feature set even slightly increases the error.

The fact that $\{f_i^1\}$ is the most important feature, and that other features only slightly improve the estimation quality, is evident from all our tests throughout the paper. To explain this, recall that since $D_i = d_i + f_i^0$, estimating D_i can be reduced to estimating f_i^0. Various works discuss the relation between f_i^0 and f_i^j values. For example, the Good-Turing frequency estimation technique [18] claims that $\frac{f_i^1}{n}$ is a consistent estimator of f_i^0.

As described in Section V-B, *effective_sampling_rate* is our metric for quantifying the computational cost of our framework. To examine the trade-off between *sampling_rate* and *training_rate*, we fix *effective_sampling_rate* to 0.02 (i.e. 2% of the stream packets are processed), and vary the *sampling_rate* between 0.005 and 0.015. We then set *training_rate* such that the *effective_sampling_rate* will be 0.02. Figure 10 shows the MAPE of this experiment with 3 batch sizes when PA is used as the online ML algorithm.

It is evident from Figure 10 that bigger batches yield better accuracy than smaller batches: as the batch size increases, the sample becomes more statistically representative, and the correlation between f_i^1 and the real batch cardinality increases.

We can also see that it is better to use a high sampling rate with a low training rate rather than vice versa. This is because the cardinality of this trace only changes slightly over time. Hence, more frequent training does not improve the accuracy over time, and it is better to invest packet processing resources in sampling more packets in every batch rather than in more training batches.

	RMSE	MAE	MAPE	MAXAE
GEE	13,803.0	13,793.6	60.9	17,441.7
AE	18,789.0	18,780.2	82.9	22,878.7
UJ2A	13,449.6	13,347.0	58.9	18,629.6
SGD	703.7	550.7	2.4	2,872.7
PA	751.3	585.0	2.6	2,798.0
RLS	704.0	549.2	2.4	2,956.7

Fig. 8: Different error metrics of the various sampling-based and ML estimators for the CAIDA-2016 trace. The algorithms used in our framework perform significantly better

E. The CAIDA-DDoS Trace

CAIDA-DDoS trace [5] contains approximately one hour of anonymized traffic from a Distributed Denial-of-Service (DDoS) attack that occurred on August 4th, 2007. We use 26,760,675 packets from this trace. These packets represent approximately 300 seconds, and they include the transition from "no-attack" to "attack". Since DDoS detection is a time sensitive application, our *estimation_rate* is a function of time rather than number of packets.

After the attack begins, the flow size distribution is heavy-tailed. Thus, as for the CAIDA-2016 trace, we expect a relatively high error rate when statistical sampling-based algorithms are used. Figure 11 shows the real cardinality vs. the cardinality estimated by statistical sampling-based algorithms, with *batch_size* of 1 second and sampling rate of $q = 0.0199$.

Next, we use CAIDA-DDoS to test our framework with different online ML algorithms and *batch_size* of 1 second. We set *sampling_rate* $= 0.01$, *training_rate* $= 0.01$, and use only

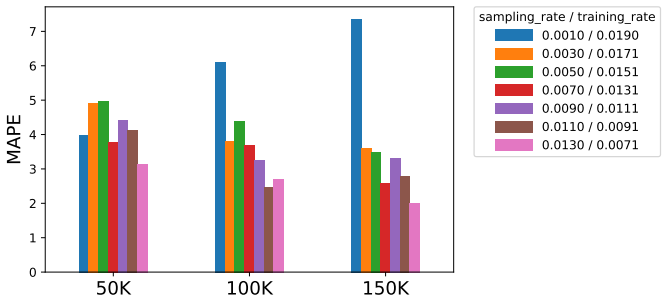


Fig. 10: The trade-off between *sampling_rate* and *training_rate* for the CAIDA-2016 trace, with fixed *effective_sampling_rate* of 2%

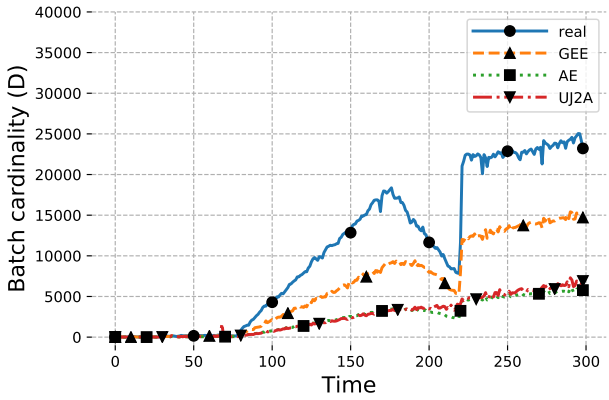


Fig. 11: Real vs. estimated batch cardinality of statistical sampling-based algorithms for the CAIDA-DDoS trace

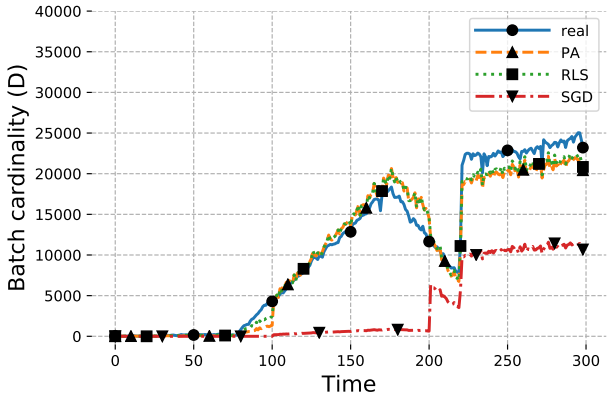


Fig. 12: Real vs. estimated batch cardinality of the proposed framework with different online ML algorithms for the CAIDA-DDoS trace

$\{f_i^1\}$ as our feature set. These parameters yield an effective sampling rate of 0.0199, which is the same as that used for the sampling-based algorithms in Figure 11. The ML-specific parameters are similar to those used for Figure 7: SGD has a *learning_rate* of 10^{-6}, RLS has a forgetting factor of $\mu = 0.99$, PA has an ϵ-insensitive loss function of $\epsilon = 0.1$, and the PA-II update rule has an aggressiveness parameter of $C = 1$. Figure 12 describes the real and estimated cardinality for each online ML algorithm.

	RMSE	MAE	MAPE	MAXAE
GEE	5,976.9	4,667.7	41.8	10,422.8
AE	11,004.0	8,539.6	75.1	19,141.3
UJ2A	10,774.4	8,362.3	73.8	18,240.5
SGD	9,548.2	7,623.9	82.9	17,474.7
PA	1,689.7	1,260.7	25.4	4,024.1
RLS	1,485.2	1,100.2	21.4	4,043.9

Fig. 13: Different error metrics of the various sampling-based and ML estimators for the CAIDA-DDoS trace

Figure 13 summarizes the error rates of Figure 11 vs. 12. We can conclude that SGD's inherent *learning_rate* is too low. Thus, it fails to adapt fast enough to changes in flow size distribution. But when we increase its *learning_rate* to 10^{-5}, the estimation diverges after the attack starts and shows even higher error rates. SGD is known for its *learning_rate* sensitivity, and it is therefore not recommended when cardinality values are expected to change drastically. In contrast to SGD, PA and RLS are significantly better than the statistical sampling-based algorithms and we see 50% improvement in the MAPE of the best online ML algorithm (RLS), over the best statistical sampling-based algorithm. As for the first trace, also in this case we do not see significant impact by extending the feature set beyond f_i^1.

Next, we examine the trade-off between *sampling_rate* and *training_rate*. We set a fixed *effective_sampling_rate* of 0.02 (2%), and use sampling rates of 0.005 (0.5%), 0.01 (1%) and 0.015 (1.5%). The training rate is calculated such that the *effective_sampling_rate* remains 0.02. Figure 14 shows the MAPE for the various algorithms for three batch sizes. It is evident that increasing the training rate while decreasing the sampling rate yields better accuracy. This is in contrast to what we saw in Figure 10 for the CAIDA-2016 trace. In the CAIDA-DDoS trace, the flow size distribution changes frequently. Hence, a higher *training_rate* is required to obtain high accuracy.

So far we have implicitly assumed that the results of each training batch are ready before the processing of the next regular (non-training) batch. However, sometimes this assumption does not hold due to the processing overhead imposed by the learning phase. To study the effect of "delayed learning" on the performance of the various algorithms, Figure 15 shows the MAPE as a function of the number of batches elapsed between the learning phase and the time when the results of this phase are available to the algorithm. The parameters used in this experiment are similar to those used for Figure 18: the sampling rate is 0.01, the training rate is 0.01, and the feature set is ['f1']. It is evident from Figure 15 that when delayed batches is 1, we get the same MAPE values as in Figure 18. It is also evident that both RLS and PA are only slightly affected

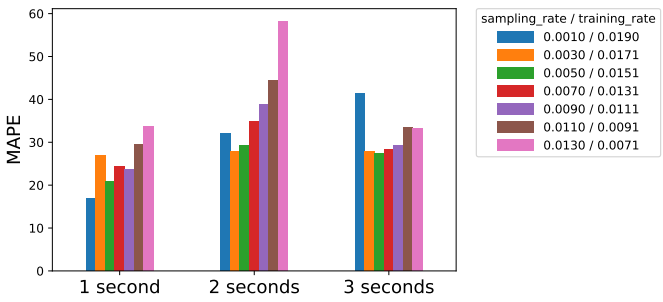


Fig. 14: The trade-off between *sampling_rate* and *training_rate* for the CAIDA-DDoS trace, with fixed *effective_sampling_rate* of 2%

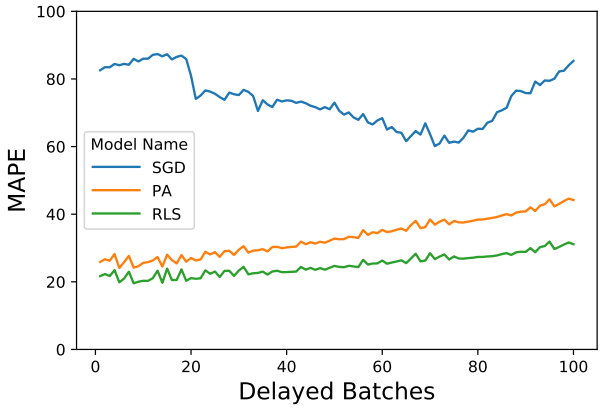


Fig. 15: The impact of delayed batches on the performance for the CAIDA-DDoS trace

by this delay, as long as it does not exceed 40, and that SGD is unstable with respect to this parameter.

F. The DARPA-DDoS Trace

The DARPA-DDoS trace [10] contains a SYN flood DDoS attack on one target (IP address 172.28.4.7) and some background traffic. The DDoS traffic comes from about 100 different sources, some of which contribute significantly more traffic than the others.

We use a part of the trace in which the attack occasionally goes on and off. It contains 3,955,270 packets, collected during 352 seconds. Since the number of packets is relatively small, a sampling rate of 1% or 2% yields a not statistically representative *sample_size*, and very high error rates. Hence, for this trace we use higher sampling rates.

Figure 16 shows the real vs. estimated cardinality of 3 statistical sampling-based algorithms with *batch_size* of 1 second and *sampling_rate* of $q = 0.0975$. Figure 17 shows the real vs. estimated cardinality of the proposed framework with the three online ML algorithms. The framework parameters used in Figure 17 are: *batch_size* $= 1$ second, *sampling_rate* $= 0.05$ and *training_rate* $= 0.05$. These parameters yield an *effective_sampling_rate* of 0.0975, which is the same as that used for the statistical sampling-based algorithms in Figure 16. The feature set consists of $\{f_i^1\}$ only. As before, SGD's *learning_rate* is set to 10^{-6}, RLS's forgetting factor is set to $\mu = 0.99$, PA's ϵ-insensitive loss function is set to $\epsilon = 0.1$, and the PA-II update rule aggressiveness parameter is set to $C = 1$.

Figure 18 summarizes the error rates of the above experiments. We can see that UJ2A's MAXAE rate is much higher than that of the other statistical sampling-based algorithms. This error is obtained from batch no. 219, which presents an extremely high cardinality value. Overall we see an improvement of 30% in the MAPE of the best online ML algorithm (PA) over the best statistical sampling-based algorithm (UJ2A).

Figure 19 presents the accuracy of PA and RLS with different feature sets. Since this trace contains a SYN attack, feature set $\{f_i^1, \textit{syn\_count}\}$ slightly improves the MAPE.

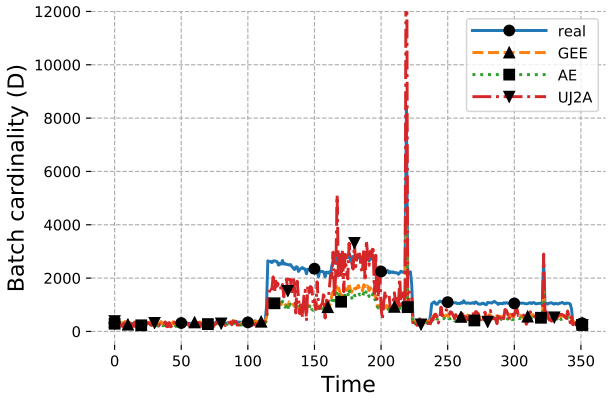


Fig. 16: Real vs. estimated batch cardinality of statistical sampling-based algorithms for the DARPA-DDoS trace

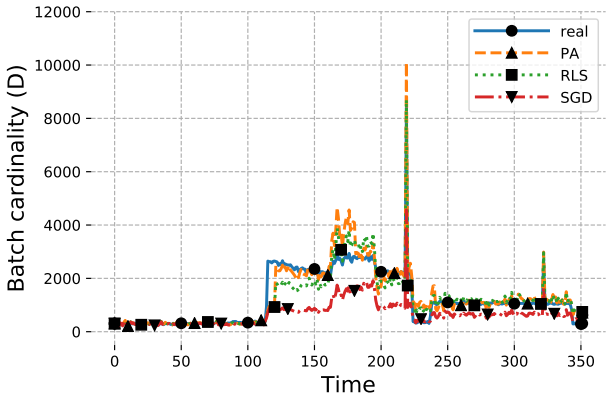


Fig. 17: Real vs. estimated batch cardinality of the proposed framework with different online ML algorithms for the DARPA-DDoS trace

	RMSE	MAE	MAPE	MAXAE
GEE	800.9	569.6	33.8	3,898.1
AE	873.4	645.0	41.4	3,729.8
UJ2A	829.4	452.0	31.2	10,826.3
SGD	447.4	271.1	23.1	1,907.2
PA	420.0	220.0	22.1	1,844.3
RLS	404.0	259.2	22.4	1,884.8

Fig. 18: Different error metrics of the various sampling-based and ML estimators for the DARPA-DDoS trace

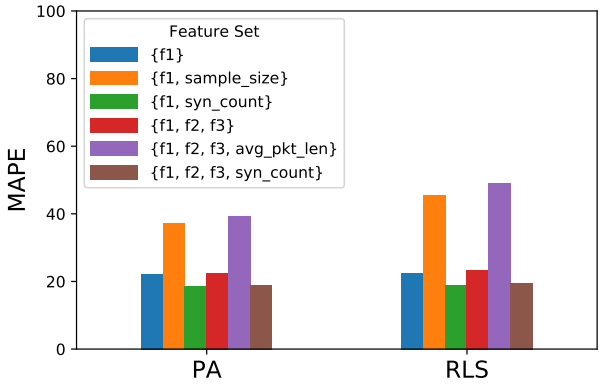


Fig. 19: The impact of extending the feature set on the accuracy of our framework for the DARPA-DDoS trace

G. The UCLA-CSD Trace

The UCLA-CSD trace [30] contains packets collected during August 2001 at the border router of the UCLA Computer Science Department. The part of this trace that we use contains 30,000,000 TCP packets, collected during approximately 14 hours.

Figure 20 shows the real vs. estimated cardinality of the three statistical sampling-based algorithms with *batch_size* of 100K packets and *sampling_rate* of $q = 0.0199$. UCLA-CSD

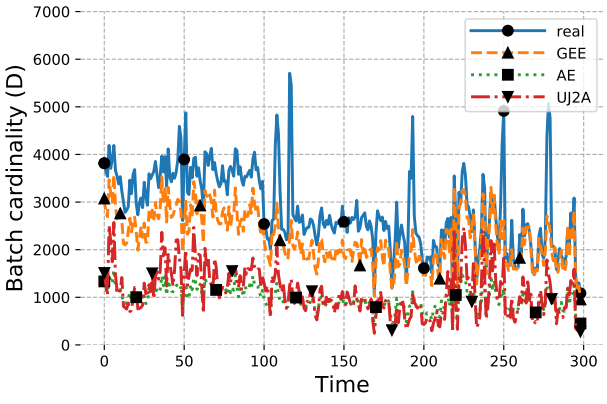


Fig. 20: Real vs. estimated batch cardinality of statistical sampling-based algorithms for the UCLA-CSD trace

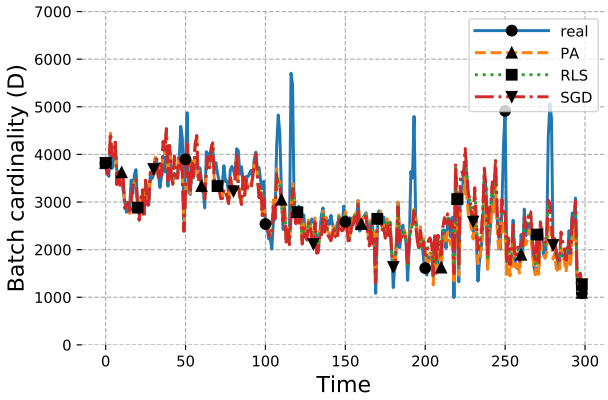


Fig. 21: Real vs. estimated batch cardinality of the proposed framework with different online ML algorithms for the UCLA-CSD trace

also shows a heavy-tailed behavior, but its skewness is less acute than that of the CAIDA-2016 trace (Section V-D). For example, analysis of the first batch shows that only 37% of the packets belong to small flows (that contain less than 4 packets). This fact can explain the performance advantage of GEE compared to AE and UJ2A.

Figure 21 shows the true cardinality vs. estimated cardinality values for our framework with each online ML algorithm. For this comparison we set the following framework parameters: *batch_size* = 100K, $q = 0.01$, *training_rate* = 0.01. The feature set contains only $\{f_i^1\}$. These parameters yield an effective sampling rate of 0.0199, which is identical to what we used in Figure 20 for the sampling-based algorithms. SGD's *learning_rate* is set to 10^{-5}, since it shows the smallest error. RLS's forgetting factor is $\mu = 0.99$, PA uses the ϵ-insensitive loss function with $\epsilon = 0.1$, and the PA-II update rule with an aggressiveness parameter of $C = 1$.

Figure 22 summarizes the error rates of the above experiments. Our framework obtains an MAPE improvement of 10% over the best statistical sampling-based algorithm. Figure 23 presents the accuracy of PA and RLS with different feature sets.

H. Computational Cost Analysis

Although calculating the estimation itself is not the most resource intensive operation, we present the following evalu-

	RMSE	MAE	MAPE	MAXAE
GEE	763.7	600.2	19.2	3,458.1
AE	1,977.4	1,863.6	64.7	4,720.7
UJ2A	1,824.0	1,735.7	61.9	4,460.2
SGD	457.0	277.2	9.7	2,972.0
PA	474.0	271.8	9.0	2,910.8
RLS	457.9	279.4	9.8	2,974.2

Fig. 22: Different error metrics of the various sampling-based and ML estimators for the UCLA-CSD trace

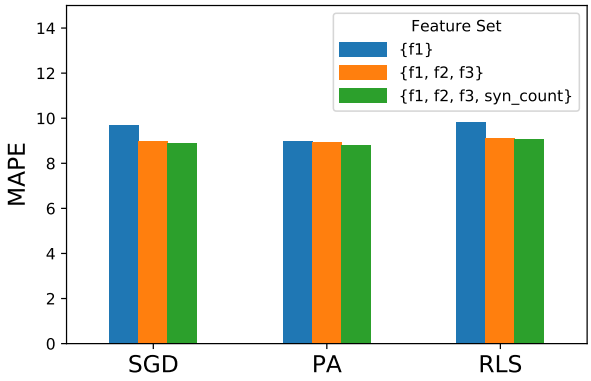


Fig. 23: The impact of extending the feature set on the accuracy of our framework for the UCLA-CSD trace

ation of its computational cost. First, we analytically analyze the runtime complexity of the computations carried out by the sampling-based algorithms, and that of the *predict()* and *partial_fit()* operations carried out by our framework. Then, we show empirical measurements to support this analysis.

Except for AE, which uses a numerical method in its estimation process and is more computationally expensive, the time complexity of all other statistical sampling-based algorithms is $O(n)$, where n is the *sample_size*, i.e., the number of packets in the sample. As for the online ML algorithms, the time complexity of each *predict()* operation is $O(1)$. The time complexity of each *partial_fit()* is $O(f^3)$ for the worst online ML algorithm (RLS), and $O(f)$ for the others (SGD and PA), where f is the number of features. Since the number of features we use is relatively small, and it does not change as more packets are processed, we expect the runtime of our framework over different *sample_size* values to be constant.

Figure 24 shows the execution time of statistical sampling-based algorithms vs. online ML algorithms, over different sample sizes. These measurements are obtained from an Intel Core i7-4500U CPU, with 8GB DDR RAM. The data contains 445 batches taken from the CAIDA-2016 trace, where each batch contains 100,000 packets. For the various online ML algorithms, we used $\{f_i^1, f_i^2, f_i^3, avg\_pkt\_len, syn\_count\}$ as the feature set. To measure CPU consumption for different *sample_size* values, we used the following sampling rates: $q = [0.1, 0.2, 0.3, 0.4, 0.5]$. Figure 24(a) shows the average execution for the sampling-based algorithms. Figure 24(b) shows the average execution time for the *predict()* operation using several online ML algorithms. Figure 24(c) shows the average execution time for the *partial_fit()* operation. As expected, in Figure 24(a) we can see the linear increase in the CPU consumption with the mean sample size, whereas in Figure 24(b) and 24(c), CPU consumption is almost constant. Finally, in Figure 24(d) we show the mean time for calculating the labels and f^i features for each batch using different

Mean Sample Size	GEE	AE	UJ2A
10,000	6.20e-06	2.72e-03	4.86e-05
20,000	7.69e-06	4.06e-03	6.99e-05
30,000	9.01e-06	5.33e-03	8.95e-05
40,000	1.03e-05	6.33e-03	1.08e-04
50,000	1.14e-05	7.44e-03	1.34e-04

(a) Average execution time in seconds for sampling-based algorithms

Mean Sample Size	SGD	PA	RLS
10,000	8.81e-06	8.52e-06	3.55e-06
20,000	7.68e-06	8.20e-06	3.62e-06
30,000	8.23e-06	8.36e-06	3.70e-06
40,000	8.15e-06	8.36e-06	3.53e-06
50,000	8.40e-06	9.25e-06	3.91e-06

(b) Average execution time in seconds of the *predict()* operation for online ML algorithms

Mean Sample Size	SGD	PA	RLS
10,000	2.38e-05	3.70e-05	2.66e-05
20,000	2.08e-05	3.49e-05	2.66e-05
30,000	2.09e-05	3.49e-05	3.06e-05
40,000	2.13e-05	3.51e-05	2.65e-05
50,000	2.82e-05	3.52e-05	2.62e-05

(c) Average execution time in seconds of the *partial_fit()* operation for online ML algorithms

Sampling Rate	Featurization (mean)	Labeling (mean)
0.1	0.05	0.45
0.2	0.10	0.48
0.3	0.13	0.42
0.4	0.19	0.46
0.5	0.21	0.41

(d) Average execution time in seconds for calculating batch cardinality and f^i features from the sample

Fig. 24: Execution time analysis for CAIDA-2016 trace.

sampling rates. The labeling time entails hashing all 5-tuples in one batch (100,000 5-tuples) and calculating the number of unique flows for that batch. The featurization time includes hashing all 5-tuples in the sample (100,000 $\cdot$ *sampling_rate* 5-tuples), calculating the number of occurrences for each 5-tuple, and calculating f^i features for the sample[3]. We can see that the time for calculating the features is proportional to the sampling rate, while the time for calculating the labels is constant.

VI. Conclusions

In this work we argued that the problem of current sampling-based algorithms for flow cardinality estimation is

[3]The time needed to load 5-tuples into memory is not included in the measurement. The calculations were done using Python's Numpy library [31] and the xxhash function [8].

their inability to adapt to changes in flow size distribution. Hence, we suggested using online ML framework to add adaptivity to the estimation process, and presented a novel sampling-based adaptive cardinality estimation framework. We analyzed the various possible features, parameters and online ML algorithms for our framework, and proposed the most suitable combination. We tested our framework over real traffic traces, and showed significant improvement in accuracy compared to the best known sampling-based algorithms while using the same amount of computational resources.

REFERENCES

[1] O. Alipourfard, M. Moshref, and M. Yu. Re-evaluating measurement algorithms in software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM, 2015.

[2] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research*. ACM, 2018.

[3] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016*.

[4] Center for Applied Internet Data Analysis - CAIDA. The CAIDA UCSD anonymized internet traces 2016 - equinix-chicago. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[5] Center for Applied Internet Data Analysis - CAIDA. The CAIDA UCSD "DDoS attack 2007" dataset. http://www.caida.org/data/passive/ddos-20070804_dataset.xml.

[6] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART*. ACM, 2000.

[7] R. Cohen, L. Katzir, and A. Yehezkel. Cardinality estimation meets Good-Turing. *Big Data Research*, 9, 2017.

[8] Y. Collet. xxhash: Extremely fast hash algorithm. http://cyan4973.github.io/xxHash/, 2016.

[9] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7(Mar), 2006.

[10] Defense Advanced Research Projects Agency - DARPA. DARPA scalable network monitoring (SNM) program traffic, PREDICT ID: USC-LANDER\DARPA_2009_DDoS_attack-20091105\rev4383. http://www.darpa2009.netsec.colostate.edu/.

[11] V. Deolalikar and H. Laffitte. Extensive large-scale study of error in samping-based distinct value estimators for databases. *arXiv preprint arXiv:1612.00476*, 2016.

[12] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[13] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2003.

[14] G. Einziger, M. C. Luizelli, and E. Waisbard. Constant time weighted frequency estimation for virtual network functionalities. In *ICCCN*. IEEE, 2017.

[15] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*. Discrete Mathematics and Theoretical Computer Science, 2007.

[16] W. A. Gale and G. Sampson. Good-Turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3), 1995.

[17] P. B. Gibbons. Distinct-values estimation over data streams. In *Data Stream Management*. Springer, 2016.

[18] I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4), 1953.

[19] I. H. GRANT. Recursive least squares. *Teaching Statistics*, 9(1), 1987.

[20] P. J. Haas and L. Stokes. Estimating the number of classes in a finite population. *Journal of the American Statistical Association*, 93(444), 1998.

[21] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. Quic: A UDP-based secure and reliable transport for HTTP/2. *IETF, draft-tsvwg-quic-protocol-02*, 2016.

[22] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013.

[23] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 1952.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16.

[26] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *ACM SIGMOD 2006*.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct), 2011.

[28] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of Open vSwitch. In *NSDI*, 2015.

[29] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[30] University of California, Los Angeles, CS department. UCLA CSD packet traces. https://lasr.cs.ucla.edu/ddos/traces/.

[31] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.