Learning-Based Heavy Hitters and Flow Frequency Estimation in Streams

Rana Shahout
Harvard University

Michael Mitzenmacher
Harvard University

***Abstract*—Identifying heavy hitters and estimating the frequencies of flows are fundamental tasks in various network domains. Existing approaches to this challenge can broadly be categorized into two groups, hashing-based and competing-counter-based. The Count-Min sketch is a standard example of a hashing-based algorithm, and the Space Saving algorithm is an example of a competing-counter algorithm. Recent works have explored the use of machine learning to enhance algorithms for frequency estimation problems, under the algorithms with prediction framework. However, these works have focused solely on the hashing-based approach, which may not be best for identifying heavy hitters.**

In this paper, we present the first learned competing-counter-based algorithm, called LSS, for identifying heavy hitters, top k, and flow frequency estimation that utilizes the well-known Space Saving algorithm. We provide theoretical insights into how and to what extent our approach can improve upon Space Saving, backed by experimental results on both synthetic and real-world datasets. Our evaluation demonstrates that LSS can enhance the accuracy and efficiency of Space Saving in identifying heavy hitters, top k, and estimating flow frequencies.

I. Introduction

The paradigm of learning-augmented algorithms, also known as algorithms with predictions, combines machine learning and traditional algorithms. This approach aims to enhance algorithms and data structures by leveraging predictions from machine learning models. Such learning-augmented algorithms have demonstrated theoretical and empirical benefits across numerous areas, including scheduling [1]–[4], caching [5], [6], and approximate frequency estimation [7]. In the realm of approximate frequency estimation, the goal is to utilize predictive models to improve the accuracy of the estimation.

A stream of packets flowing through a network or a specific switch can be divided into multiple flows, each identified by a unique set of attributes. For example, the 5-tuple consists of source IP, source port, destination IP, destination port, and protocol is often used as a flow identifier. Two fundamental problems in network statistics are determining the frequency of a flow, i.e., counting the number of packets with a given flow identifier that passes through the network or switch, and identifying the heavy hitter flows (the most frequent ones), as these flows often have the most significant impact. These monitoring capabilities are desirable for various network algorithms and domains, such as load balancing [8], routing, fairness [9], intrusion detection [10], [11], caching [12], and policy enforcement [13].

In a modern network, as the number of flows can be massive, providing precise answers to queries regarding flow[1] frequencies and identifying heavy hitters[2] is often prohibitively costly. To that end, streaming algorithms process the data in a single pass while efficiently estimating flow frequencies with minimal storage space are essential. These algorithms build a *sketch*; a compact data structure that extracts statistical information in one pass on the entire data.

Standard frequency estimation algorithms can be broadly classified into two categories: hashing-based and competing-counter-based approaches. The hashing-based approach [14]–[16] hashes data items into buckets, potentially leading to collisions among different items. It then calculates item frequencies based on the number of items hashed into each bucket. In contrast, the competing-counter-based approach, such as Space Saving [17], maintains a fixed number of counters that at any time tracks a subset of the input items, without hashing and corresponding collisions. These data stream items "compete" for the limited counters, with the algorithm aiming to allocate counters to the most frequent items. For identifying heavy hitters and top k, while hashing-based approaches require additional structures like heaps, competing-counter-based approaches do not necessitate such auxiliary components, making them more space-efficient. Indeed, competing-counter-based algorithms have been shown to be more space-efficient than hashing-based algorithms, both empirically and asymptotically [18], [19], as hashing-based algorithms require allocating significantly more counters and space than theoretical lower bounds to mitigate the impact of hash collisions on counter accuracy. Additionally, competing-counter-based algorithms offer deterministic error guarantees, in contrast to hashing-based, which are randomized.

Hsu et al. [7] introduced a learned hashing-based algorithm for frequency estimation that utilizes a heavy hitter predictor. When an incoming item is predicted to be a heavy hitter, it is assigned a unique bucket for accurate counting. Items not predicted as heavy hitters are processed using a conventional sketching algorithm. This approach reduces collisions between heavy hitters and less frequent items, leading to improved overall accuracy. It is important to note that heavy hitter

[1]The terms flow and item are used interchangeably.

[2]The terms heavy hitters and frequent items are used interchangeably.

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

predictors may exhibit errors, and thus relying solely on them to identify heavy hitters can violate error guarantees. To our knowledge, there is a lack of research on learned competing-counter-based algorithms. Unfortunately, directly applying the same strategy to competing-counter-based algorithms by simply assigning all counters to predicted heavy hitters leads to unbounded estimation error. For instance, a falsely predicted heavy hitter item occupies a counter that could have been used for a real heavy hitter. (Recall that competing-counter-based approaches have a fixed number of counters.) Even worse, a true heavy hitter might be mistakenly identified as a non-heavy hitter and consequently ignored. Unlike learned hashing-based algorithms that focus only on heavy-hitter items, we propose an approach that employs a predictor to filter out noise caused by items predicted to have low frequencies as in many network traces [20] and practical distributions (e.g. Zipfian), many items have only one occurrence.

In this paper, we introduce a learned competing-counter-based algorithm for identifying heavy hitters, top k, and flow frequency estimation. We focus on the Space Saving algorithm, although our approach could be applied to any competing-counter-based algorithm, such as the MG algorithm [21]. We present Learned Space Saving (LSS), a novel technique that leverages machine learning predictions to guide the competition for limited counters among data items. Specifically, LSS aims to exclude predicted "weak" or low-frequency items from the competition, while ensuring that predicted "strong" or heavy-hitter items remain in the competition. In this way, the Space Saving accuracy is improved.

LSS is designed to be resilient against prediction errors. Our LSS method employs a filtering mechanism to exclude predicted low-frequency items, resulting in a "cleaner" data structure. To ensure robustness against incorrectly predicting a high-frequency item as a low-frequency item, we employ a Counting Bloom filter that tracks these items and ensures an item is not consistently ignored just because it is (repeatedly) predicted as low frequency. When using the heavy hitter predictor, we divide the counters into fixed and mutable counters. A predicted heavy hitter can be allocated a fixed counter, to achieve an accurate count. But fixed counters are limited, so if there are excessive incorrect heavy hitter predictors, the mutable counters allow for a standard Space Saving algorithm on items after the fixed counters are filled. We break LSS into two variants, LSS-LF and LSS-HH, each utilizes a distinct learning model and is designed to exhibit resilience against prediction errors.

Our contributions are: 1) We propose LSS, a learning-based approach for identifying heavy hitters, top k and frequency estimation, improving Space Saving's accuracy and designed to be robust against prediction errors (Section III). 2) We break down LSS into LSS-LF (Section IV) and LSS-HH (Section V), showing each component's robustness. 3) We present theorems providing insight into potential gains 4) We propose LSS+, a variation of LSS offering higher update throughput by relaxing deterministic to probabilistic approximation guarantees (Section VI). 5) We implement and evaluate LSS and variants (Section VII) on synthetic and real-world datasets (Internet traffic, web search); LSS demonstrated up to 18% better top-k precision, 24% higher heavy hitter recall, and 34% lower RMSE for frequency estimation compared to Space Saving under certain configurations.

II. Background

A. Preliminaries

Given a *universe* $\mathcal{U}$, a *stream* $\mathcal{S} = u_1, u_2, \ldots \in \mathcal{U}^N$ is a sequence of arrivals from the universe. (We assume the stream is finite here.) We define the frequency of an item i in $\mathcal{S}$ as the number of items corresponding to i in $\mathcal{S}$ and denote this quantity by f_i.

We seek algorithms that support the following operations:

- **ADD(i)**: given an element $i \in \mathcal{U}$, append i to $\mathcal{S}$.
- **QUERY(i)**: return an estimate $\widehat{f_i}$ of f_i

A weighted stream consists of tuples of the form (u_i, w_i), where u_i represents the item's id and w_i its (non-negative) weight. At each step, a new tuple is added to the stream. In this setting, the weight w_i is added to the corresponding item's frequency. Frequency estimation algorithms typically assume unweighted updates, such as click streams, while others assume weighted updates, such as network traffic volumes. In this paper, we focus on the unweighted updates model for ease of exposition, although our approach applies to weighted data streams as well, with straightforward modifications.

Definition 1. *An algorithm solves ϵ-Frequency if given any Query(i) it returns $\widehat{f_i}$ satisfying*

$$f_i \leq \widehat{f_i} \leq f_i + N\epsilon.$$

A (randomized) algorithm is said to solve ϵ-Frequency with probability $1-\delta$ if, for any i chosen before the stream is processed, Query(i) returns $\widehat{f_i}$ satisfying the above bound with probability $1-\delta$. Similarly, a (randomized) algorithm solves ϵ-Frequency in expectation if given any Query(i) it returns $\widehat{f_i}$ satisfying

$$f_i \leq E[\widehat{f_i}] \leq f_i + N\epsilon.$$

Definition 2. *An algorithm solves (ϵ, θ)-HeavyHitters if it returns a set of items B, such that for every item i: if $f_i \geq \theta N$, then $i \in B$, and if $f_i \leq (\theta - \epsilon)N$ then $i \notin B$. An algorithm solves (ϵ, θ)-HeavyHitters with probability $1-\delta$ if it returns a set of items B satisfying the above conditions with probability $1-\delta$.*

Deterministic solutions (competing-counter-based) [17], [21] for the ϵ-HeavyHitters problem ensure the identification of all items with sufficiently large counts, but potentially may include some items with counts smaller than the given heavy hitter threshold. In contrast, hashing-based [14], [15] for the ϵ-HeavyHitters problem may introduce a probability of failure.

B. Robustness in Learning-Augmented Algorithms

As mentioned, a learned-augmented algorithm combines traditional algorithms with machine learning models. (It should be noted, however, that this approach generally treats the

machine learning models as black boxes, allowing it to work with any model that provides usable predictions.) However, machine learning methods are inherently imperfect and may exhibit errors, including substantial and unexpected errors. A key question is how can we use predictions while maintaining robustness, which refers to ability of an algorithm to maintain reasonable performance even if the predictions are simply wrong [5], [22]. Ensuring robustness is essential because machine learning models are rarely perfect in practice. There are several reasons why learned models may exhibit errors. First, most models are trained to perform well on average by minimizing expected losses. In doing so, they may reduce errors on the majority of inputs at the expense of increasing errors on outlier cases. Additionally, generalization error guarantees only hold when the training and test samples are drawn from the same distribution. If this assumption is violated, due to distribution drift or adversarial samples, the predictions can vary from the ground truth.

One general approach for learned-augmented algorithms to try to achieve robustness is to fall back on the traditional algorithm when the model is inaccurate. This requires being able to notice inaccurate predictions and change to the traditional algorithm quickly and effectively. Another related approach, which we use here, is to find ways to limit the damage that can be caused by incorrect predictions by using additional algorithm or data structure.

C. Space Saving

The Space Saving (SS) algorithm [17] is a competing-counter algorithm that provides frequency estimation for data stream items. Space Saving maintains a set of k entries, denoted by T, each entry has an associated item i and counter, and we use c_i to denote the counter value associated with item i if any exists. When $k = \frac{1}{\epsilon}$, Space Saving estimates the frequency of any item with an additive error less than $N\epsilon$ where N is the stream size.

When an item i is encountered within the stream that is in the set T, the algorithm increments its corresponding counter c_i. In cases where the item i is not present in T and the size of T is less than k, the algorithm adds i to the set T and initializes its count to 1 ($c_i = 1$). Otherwise, when the item i is not in T and the size of T has reached k, Space Saving identifies an item j within T with the minimum non-zero count c_j, denoted by $minCount$. The algorithm then executes an update in which c_i is assigned the value of $c_j + 1$, and the set T is updated to replace item j with item i. To estimate the frequency of an item, if the item is in T then we report its count. Otherwise, we report the smallest counter stored in T. Pseudocode for SS is presented (in black) in Algorithm 1.

Space Saving satisfies the following properties (the first two properties are proved in [17] while the latter is proved in [23]):

Lemma 1. *Space Saving with $k = \epsilon^{-1}$ counters ensures that after processing N insertions, the minimum count of all monitored items is no more than $\frac{N}{k} = N\epsilon$, i.e, $minCount < N\epsilon$.*

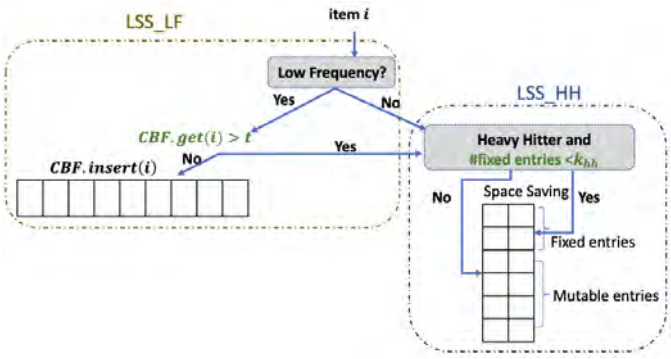


Fig. 1: Overview of LSS, combining the approaches of LSS-LF and LSS-HH. LSS-LF utilizes a low-frequency predictor to exclude infrequent items. LSS-HH divides entries into fixed and mutable entries, using a heavy hitters predictor to allocate fixed entries for frequent items while processing remaining items through the mutable entries. To mitigate the impact of prediction errors, LSS-LF utilizes a Counting Bloom Filter (CBF), while LSS-HH sets a limit (k_{hh}) on the number of fixed entries (represented in green color).

Lemma 2. *All items with frequency larger than or equal to $minCount$ are in the set T, the set of items with associated counters.*

Lemma 3. *Space Saving with ϵ^{-1} counters can estimate the frequency of any item with an additive error less than $N\epsilon$.*

III. Learned Space Saving

We propose a learning-based approach to enhance the accuracy of the Space Saving algorithm, called Learned Space Saving (LSS). Our approach allows for two types of predictors: one for identifying low-frequency items and another for identifying heavy hitters. Either or both can be used. Alternatively, a single stronger predictor capable of predicting item frequencies could be employed. To accommodate the availability of predictors in different systems, we divide LSS into two components: LSS with low frequency predictor (LSS-LF) when only the low-frequency predictor is available, and LSS with heavy hitter predictor (LSS-HH) when only the heavy hitters predictor is available. When both predictors are available, or when a single predictor can predict item frequencies, we utilize the full LSS, which combines the strengths of the individual components.

LSS-LF employs a filtering mechanism to exclude low-frequency items, resulting in a "cleaner" data structure. It utilizes a Counting Bloom Filter to ensure robustness against prediction errors. LSS-HH takes a different approach by dividing the Space Saving counters into two categories: fixed counters and mutable counters. It uses a heavy hitters predictor to allocate fixed counters for items identified as heavy hitters. The remaining items are processed using the traditional Space Saving algorithm, using mutable counters for maintaining their counts. This overall approach is illustrated in Figure 1.

Algorithm 1 LSS

Initialization: $T \leftarrow \varnothing$, $T_{mutable}$– *Mutable entries in* T
CBF– Counting Bloom filter, t– CBF threshold
freqPredictor – frequency predictor
$fixedEntries \leftarrow 0$
HH - heavy hitters predictor, k_{hh} - number of allowed fixed entries
function INSERT(i)
  if *freqPredictor(i)* $> t$ or $CBF.GET(i) > t$ **then**
    if $i \in T$ **then**
      $c_i \leftarrow c_i + 1$
    else
      if $|T| < k$ **then**
        $T \leftarrow T \cup \{i\}$
        $c_i \leftarrow 1$
      else
        $j \leftarrow \text{argmin}_{j \in T_{mutable}} c_j$
        $c_i \leftarrow c_j + 1$
        $T \leftarrow T \cup \{i\} \setminus \{j\}$
      end if
      if HH(i) and $fixedEntries \leq k_{hh}$ **then**
        mark entry of c_i as fixed entry
        $fixedEntries \leftarrow fixedEntries + 1$
      end if
    end if
  else
    $CBF.ADD(i)$
  end if
end function
function QUERY(i)
  if $i \in T$ **then**
    return $c_i + t$
  else
    return $\min_{j \in T} c_j + t$
  end if
end function

Our design decouples the robustness logic (highlighted in green in Figure 1) from the remaining algorithm logic. Figure 2 illustrates the algorithms presented in this paper. We have employed a color-coding scheme to aid readers in visualizing and distinguishing between the presented algorithms. We represent four algorithms in the pseudo-code of Algorithm 1, each uniquely represented using specific colors, as follows: Space Saving is represented in black. LSS-LF combines black and brown colors. LSS-HH combines black and blue colors. Finally LSS combines all of them.

IV. LSS-LF: Learned Space Saving without Low Frequencies

We claim that by selectively removing low-frequency items and properly adjusting their estimation, we can enhance the accuracy of the Space Saving algorithm. We first consider the case of items appearing only once in a stream and "remembering" their count as 1 as a special case. Then, we present the general case of removing low-frequency items up to t occurrences.

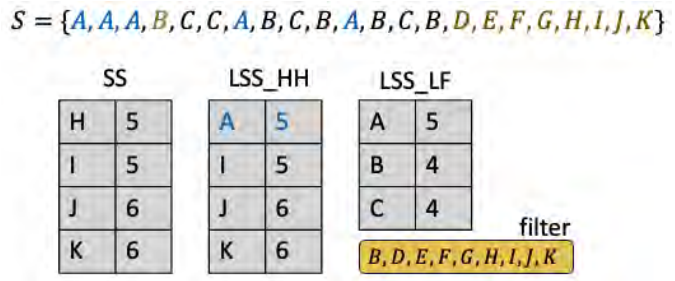


Fig. 2: An illustration of our algorithms. Consider an input stream S of 11 items as shown, where A is predicted as a heavy hitter, the first arrival of B is predicted (incorrectly) as low frequency, D, E, F, G, H, I, J, K are predicted (correctly) as low-frequency items. In this example, low-frequency items take over the SS counters. In LSS-HH, by allocating a fixed entry for the predicted heavy hitter A, the remaining counters again are taken over by low-frequency items. LSS-LF, however, ensures that low-frequency items do not dominate by filtering them. When B arrives after being previously stored in the filter, it is tracked again. Note that LSS-LF uses fewer counters than SS and LSS-HH to account for the additional memory required for the filter.

Our intuition is that occurrences of items that appear once "disturb" the accuracy of items that are in T. Space Saving could set a counter of an item that is potentially frequent (i.e. heavy hitter) to zero by encountering a single occurrence item. Eventually, the replaced frequent item will return to the Space Saving table (from correctness), but in this case, the "counter history" will be lost, causing inaccuracy in the counters.

We accordingly introduce LSS-LF (Learned Space Saving without Low Frequencies), which aims to exclude items that are predicted to have up to t occurrences from being inserted into Space Saving. We refer to the special case when $t = 1$ by LSS-LFS (Learned Space Saving Low Frequency Singles).

A. Addressing Items with Single Occurrence

We focus on this particular case for two primary reasons: First, for some distributions (see Figure 3), the majority of low-frequency items are items that occur only once. Second, in certain setups (e.g. sliding windows [24]), obtaining a predictor for single occurrence items is more achievable than a general low-frequency predictor. Even by addressing this special case of single occurrences, we demonstrate improved performance.

LSS-LFS employs a predictor that, for a specific item i, predicts if item i has a single occurrence. Note that we perform predictions for every incoming arrival.

Unless some mitigating structure is added, ignoring predicted single items can lead to unbounded errors. For example, if a heavy hitter is predicted incorrectly as a single item, this item is excluded from the SS, violating the error guarantee.

We therefore add a structure to ensure that when the predictor suggests an item be ignored, a verification process is used to determine if the item has been previously ignored, and ignores the prediction if that has occurred. Our approach uses a Bloom Filter (BF) [25], [26][3] (we consider further generalizations later). A BF is an efficient probabilistic data structure that checks if an item is in a set, allowing false positives but no false negatives.

Specifically, we keep a Bloom filter of predicted single occurrence items previously ignored to ensure robustness. If the predictor predicts an item is a single occurrence, but the Bloom filter returns that it has been previously ignored, this indicates the item is not a single occurrence item, we disregard the predictor's suggestion and instead insert the item into the Space Saving. Otherwise, if the item is not found in the Bloom filter, we add it to the filter and ignore it. For example, in Figure 2, the items $B, D, E, F, G, H, I, J, K$ are predicted as low-frequency items and are eliminated from being inserted into the table. Instead, they are placed in the filter. B is tracked again as it was incorrectly predicted as a low frequency item. As a result, the counters remain dedicated to tracking non-low-frequency items.

The effect is that faulty predictions will not cause us to ignore an item more than once, because Bloom filters are designed so that there are no false negatives. However, we may underestimate the count of mispredicted items by 1, because an item is not included in the Space Saving when it is inserted in the Bloom filter. We compensate for this by adding 1 to the query result of the Space Saving. This approach differs from the strawman method that uses a Bloom filter before Space Saving to filter out infrequent items. The strawman's Bloom filter includes all distinct items in the stream, leading to higher memory usage. In LSS-LFS, leveraging predictions, the filter contains only items the predictor suggests ignoring.

1) Robustness Result: We first present theoretical results showing that LSS-LFS is robust, in the sense that it cannot behave too much worse than the corresponding algorithm that does not use predictions (denoted by SS).

Theorem 1. *Let SS be an algorithm for $(\epsilon - \frac{1}{N})$-Frequency. Then LSS-LFS (Algorithm 1) solves ϵ-Frequency.*

Proof. For any item i and stream size N, we have

$$\widehat{f}_i \triangleq SS_{(\epsilon-\frac{1}{N})}.\textsc{Query}(i) + 1. \tag{1}$$

That is, our estimate is the query result for i from SS, which is an algorithm for $(\epsilon - \frac{1}{N})$-Frequency, with at most one occurence of i removed from the stream SS processes and an extra count of 1 added back in. It follows the smallest possible return value is $(f_i - 1) + 1 = f_i$, and the largest possible return value is $\left(f_i + \left(\epsilon - \frac{1}{N}\right) N\right) + 1 = f_i + \epsilon N$, proving the claim.

Note that we alternatively could have used a ϵ-Frequency algorithm for SS and not added 1 to the query result, yielding a largest possible return value $\widehat{f}_i$ of $f_i + \epsilon N$, but might yield a smallest value of $f_i - 1$ (as a lower bound). We chose the presentation of Theorem 1 to maintain the same guarantees as without predictions. □

Theorem 2. *Given $k = \epsilon^{-1}$ available counters and the availability of perfect predictions for items with a single occurrence, let ℓ represents the count of such items. By utilizing this predictor specifically to filter out single-occurrence items, Space Saving can estimate the frequency of any item with additive error less than $(N - \ell)\epsilon$, where N denotes the size of the stream.*

Proof. In the Space Saving algorithm, the minimal counter value is always greater than or equal to the ratio of the number of inserted items to the number of counters. Since we eliminate items with single occurrences, the total number of inserted items becomes $N - \ell$. By using ϵ^{-1} counters, we ensure that the minimal counter value is greater than or equal to $(N-\ell)\epsilon$. The rest of the proof follows immediately from Lemma 1, 3. □

Our LSS-LFS algorithm will, of course, not be as good as Theorem 2 even with perfect predictions, because it does not assume predictions are perfect, and uses a Bloom filter for robustness. However, the following theorem provides insight into why we expect strong benefits, given suitably good predictors. In stating the theorem, we recall that a Bloom filter has a false positive rate, which corresponds to the probability a non-set item creates a false positive, and further, for any sequence of M Bloom filter queries, the fraction of false positives is at most $(1 + \nu)M$ for any constant ν with high probability (that is, $O(M^{-\alpha})$ for any constant α; note the result may require sufficiently large M).

Theorem 3. *Given SS with $k = \epsilon^{-1}$ to solve the ϵ-Frequency problem and given perfect predictions of whether an item is a single occurrence. LSS-LFS using k counters can estimate the frequency of any item with additive error less than $(N - \ell \cdot (1 - (1 + \nu)fpr)\epsilon$ with high probability, where fpr is the false positive rate of its Bloom filter and $\nu > 0$ can be any suitable constant.*

Proof. Based on Algorithm 1, LSS-LFS uses a predictor to filter out arrivals with a single occurrence. A perfect predictor is assumed, but without knowing that it is perfect. As LSS-LFS utilizes a Bloom filter to handle the predictor's imperfections, which may yield false positives that include unnecessary items in its Space Saving instance. The expected number of false positives can be expressed as $\ell \cdot fpr$, and with high probability the number of false positives is at most $(1+\nu)\ell \cdot fpr$. Following the same logic as in Theorems 2, it can be deduced that the total number of inserted items, with high probability, is $N - \ell \cdot (1 - (1 + \nu)fpr)$. □

B. Addressing Items Up to t Occurrences

For given t, the previous approach can be generalized by addressing items with up to t occurrences, where t is a threshold that depends on the specific distribution.

[3]The BF could be replaced with any similar filter structure, such as a cuckoo filter or ribbon filter [27], [28].

We now present LSS-LF, a generalization of LSS-LFS. We use a predictor that, for a specific item i and threshold t, predicts if item i has more than t occurrences. LSS-LF replaces the standard Bloom filter with a Counting Bloom Filter (CBF) [29] which expands on the Bloom filter capabilities by tracking how many times each item appears in a multi-set. The CBF tracks the number of times an item is inserted into it, so we can (approximately) count how many times an item that is predicted to have less than t occurrences within the stream, and then we place items into the Space Saving if their CBF count reaches a predefined threshold t. This generalizes the previous algorithm using the Bloom filter which corresponds to the case $t = 1$. Now, however, our underestimation might be as much as t, from t insertions until reaching the CBF threshold. We correct this by adding t to the value returned by SS in LSS-LF.

Theorem 4. *Given a threshold t, let SS be an algorithm for $(\epsilon - \frac{t}{N})$-Frequency. Then LSS-LF solves ϵ-Frequency.*

Proof. The proof follows the same logic as that of Theorem 1. Here, however, we may lose up to t entries for an item i within the stream. □

V. LSS-HH: Learned Space Saving with Heavy Hitters

Here, we explore the case where a predictor for heavy hitters is included. We propose LSS-HH as an approach that exploits the heavy hitter predictor for better performance. LSS-HH assigns a set number of entries specifically for predicted heavy hitters in the Space Saving algorithm. As a result, these entries are protected from replacement, even when a new item appears. The other remain mutable, as with the original Space Saving algorithm; in particular, when an item appears that replaces the item with the lowest counter value, it does so only with respect to the mutable entries. We discuss additions to this approach to ensure it remains robust in the face of potential prediction errors.

Inaccurate predictions can lead to a scenario where small items erroneously take up fixed entries and reduce the available fixed entries originally intended for actual heavy hitters. As a result, heavy hitters are subject to frequent inclusion and removal from the structure, resulting in unbounded errors. To ensure the robustness of our approach, we set a limit on the maximum number of fixed entries (k_{hh}). In cases where the number of predicted heavy hitters exceeds the defined threshold, the initial heavy hitters encountered in the data stream fill the fixed entries, while the remaining heavy hitters compete on regular mutable entries. In contrast, if the number of predicted heavy hitters is less than the fixed entries, those entries will be given to non-heavy hitters. This is due to the fact that we only mark entries as fixed when a heavy hitter shows up. In Figure 2, there is one fixed entry allocated for A, which is predicted as a heavy hitter. This prevents low-frequency items from replacing the counter tracking the heavy hitter A.

Theorem 5. *Given the availability of perfect predictions for heavy hitters, LSS-HH using k counters, where k_{hh} among them are fixed entries, provides exact frequencies for k_{hh} heavy hitters (zero errors) and estimates the frequency of the other items with additive error less than $\frac{N-k_{hh}\theta N}{k-k_{hh}}$ where θ is the heavy hitters' threshold.*

Proof. Since heavy hitters have dedicated counters, their error is zero. For the remaining items, we have $k - k_{hh}$ (mutable) counters. As before, the minimal counter value among the mutable counters is always greater than or equal to the ratio of the number of inserted items to the number of counters. However, since we eliminate heavy hitter items, each appearing at least θN times, the total number of inserted items becomes $N - k_{hh}\theta N$. Thus, the rest of the proof follows immediately from Lemma 1 and Lemma 3. □

If the frequency distribution of items is Zipfian, then the number of θ-heavy hitters is at most $\frac{1}{\theta \ln n}$, where n represents the total number of unique items in the stream (Remark 9.13 in [7]). In this case, k_{hh} can be set to $\frac{1}{\theta \ln n}$.

VI. LSS+: Faster LSS

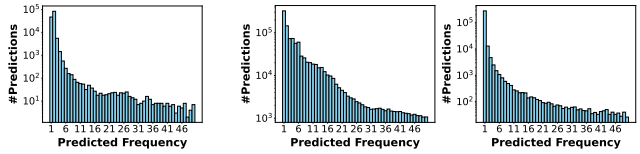
The inclusion of the filter (BF or CBF), while crucial for robustness, can potentially slow down overall performance and increase memory usage. We present LSS+ that offers potentials for speedup at the cost of loosening our deterministic approximation guarantees to probabilistic guarantees. For simplicity, we describe LSS+ for the case of filtering out single occurrences, but the approach also applies to filtering out items with up to t occurrences.

Whenever the predictor suggests an item will have a single occurrence, LSS checks for the item's presence in the Bloom filter, and either adds it to the Bloom filter if it is not there, or adds 1 to the item count in the Space Saving data structure if it is. LSS+ mitigates this overhead by selectively executing this step with probability τ, and adding τ^{-1} to the item count in the Space Saving data structure if needed. (For convenience, we assume here that τ^{-1} is an integer, to avoid floating point arithmetic, but one could develop alternative implementations.)

In the special case where τ equals 1, LSS+ coincides with LSS. When τ is less than 1, it takes on average τ^{-1} predictions that an item will have a single occurrence before checking the Bloom filter, reducing accesses to and memory required for the Bloom filter. However, now the guarantees for the algorithm are only in expectation; when an item is in the Bloom filter, its *expected* count increases by one each time such a prediction occurs. The value of the parameter τ in LSS+ can be chosen to balance between robustness and efficiency.

As LSS+ checks the Bloom filter probabilistically, it makes sense to consider the expected value of the query response; we refer to this as solving the query problem in expectation.

Theorem 6. *Let SS be an algorithm for $(\epsilon - \frac{\tau^{-1}}{N})$-Frequency. Then LSS+ solves ϵ-Frequency in expectation.*



(a) Web search (b) IP (c) Zipf $\alpha = 1.3$

Fig. 3: Histogram of predicted frequencies up to 50 of the used datasets (Log scale). For Web search and IP datasets, we use the learned model described in Section VII, for the Zipf distribution, we used the simulated predictor with $p = 0.9$.

Proof. The proof follows similar logic as that of Theorem 1. The return value of LSS+ is

$$\widehat{f_i} \triangleq SS_{(\epsilon - \frac{\tau^{-1}}{N})}.\text{QUERY}(i) + \tau^{-1}. \tag{2}$$

Each time item i is encountered, if the prediction says it will appear again in the stream, its count is increased by 1 deterministically. If the prediction says it has a single occurrence, but it is in the Bloom filter, its expected count increases by 1, as it goes up by τ^{-1} with probability τ (when the Bloom filter is checked). The only times the expected count does not increase by 1 on each appearance is when the item is not in the Bloom filter and it is predicted to be a single occurrence; this occurs an expected τ^{-1} times before the item is put in the Bloom filter. It follows that $E[\widehat{f_i}] \geq f_i$ and $E[\widehat{f_i}] \leq f_i + \left(\epsilon - \frac{\tau^{-1}}{N}\right) N + \tau^{-1} = f_i + \epsilon N$.

□

VII. Evaluation

Our evaluation examines LSS's effectiveness compared to SS using the following datasets:

- **IP Trace Datasets:** : We use the anonymized IP trace streams collected from CAIDA [20]. The traffic data is collected at a backbone link of a Tier 1 ISP between Chicago and Seattle in 2016. Each recording session lasts approximately one hour, with around 30 million packets and 1 million unique flows observed within each minute.
- **Web Search Query Datasets:** We use the AOL query log dataset [30], which comprises 21 million search queries collected from 650 thousand anonymized users over a 90-day period. The dataset contains 3.8 million unique queries, each consisting of a multi-word search phrase.
- **Synthetic Datasets:** We generated synthetic datasets following the Zipf [31] distribution with varying skewness levels, each dataset contains 10 million items.

Implementation and Computation Platform: We implemented the learned versions of Space Saving in Python 3.7.6. The evaluation was performed on an AMD EPYC 7313 16-Core Processor with an NVIDIA A100 80GB PCIe GPU, running Ubuntu 20.04.6 LTS with Linux kernel 5.4.0-172-generic, and TensorFlow 2.4.1.

The Learned Model: We follow the implementation of the learned model in [7] and adapt it using TensorFlow 2.4.1 for the discussed datasets CAIDA [20] and AOL [30]. For the CAIDA dataset, we train a neural network[4] to predict the log of the packet counts for each flow. The model takes as input the IP addresses, ports, and protocol type of each packet. We employ Recurrent Neural Networks (RNNs) with 64 hidden units to encode IP addresses and extract the final states as features. Ports are encoded by two-layer fully-connected networks with 16 and 8 hidden units. The encoded IP and port vectors are concatenated with the protocol type, and this combined feature vector is used for frequency estimation via a two-layer fully-connected network with 32 hidden units. For the AOL dataset, we construct the predictor by training a neural network to predict the number of times a search phrase appears. To process the search phrase, we train an RNN with LSTM cells that take the characters of a search phrase as input. The final states encoded by the RNN are fed to a fully connected layer to predict the query frequency. The model is trained on a subset of data to identify properties that correlate with item frequencies instead of memorizing specific items. This trained model is then tested on a separate dataset.

Simulated Frequency Prediction: For synthetic datasets, given true frequencies, we generate predicted frequencies.We simulate a predictor as follows: given a threshold t and probability p, items are classified as either small (true count $< t$) or big (true count $\geq t$). With probability $1-p$, an item is mispredicted where small (big) items are mispredicted as big (small) items, and their predicted count is randomly chosen from the set of big (small) item counts. With probability p, the prediction of an item is its true count multiplied by a factor that slightly varies around 1, within a range defined by the noise level (default 5%).In addition, for items below the threshold t, a small probability (default 1%) controls the likelihood that the added noise will cause these items to be predicted above t. For CAIDA and AOL datasets, we use the learned predictor from Section VII. When demonstrating robustness on real datasets, we employ controlled prediction with a simulated predictor to introduce non-perfect predictions.

Parameter Setting: Our approach does not aim to optimize every parameter thoroughly. In practice, parameters can be fine-tuned based on knowledge of data distribution, or by iterative refinement over time, benefiting from our scheme's inherent robustness. Here we state the default parameters we used in the experiments unless stated otherwise. For a fair comparison with the same memory consumption as SS, we allocated 90% of the memory consumed by SS counters to the LSS counters and the remaining 10% to the filter (CBF), we refer to this as filter ratio in the experiments. When using fixed counters, particularly for finding heavy hitters, we designated 10% of the total counters as fixed counters. (We also explore the effects of varying the number of fixed counters.) Since we do not assume a prior knowledge of the dataset, we set a low frequency threshold $t = 4$. To set the threshold for identifying heavy hitters, we used the theoretical error guarantee ϵ based

[4]Again, the implementation of such a predictor is our tailored approach, and it is just one of many possible options. Depending on the requirements, our design may be replaced or enhanced with any other effective prediction technique.

on the used memory (Lemma 1). The default value for this threshold is chosen to be $\theta = 0.25\epsilon$. To simulate periodic queries throughout the data stream, we execute a query at intervals of every 1000 arrivals. The reported accuracy is the average across all windows.

A. Problems and Metrics

We explored three problems: (1) detecting heavy hitters in the stream, i.e., items whose frequency exceeds a given threshold; (2) finding the top k most frequent items in the stream, where k is given; and (3) estimating the frequencies of individual items in the stream. Our error metrics are:

Root Mean Square Error (RMSE) for Frequency Estimation: measures the square root of the average squared differences between the estimated frequency and actual frequency. $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f_i - \hat{f}_i)^2}$.

Precision for Top-k: ratio of the number of correctly reported instances to the number of reported instances ($\frac{TP}{TP+FP}$), where TP is the true positive and FP is the false positive.

Recall for Heavy Hitters: ratio of the number of correctly reported instances to the number of correct instances ($\frac{TP}{TP+FN}$) where FN is the false negative.

Operations performance: insertions or queries per second.

B. End-to-End Performance

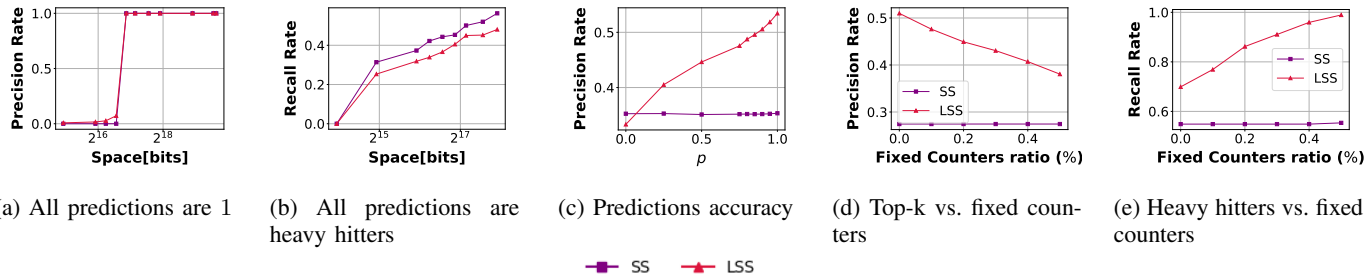
a) Data Skew: To show how the numerous less frequent tail items could collectively dominate the counters in a space-saving algorithm, we present in Figure 3 histograms for the frequency range of 1 to 50, plotted on a logarithmic scale. The scale highlights that there are a large number of low-frequency items. We observe a sharp peak at the lowest frequency (1) and as the predicted frequencies increase, the number of items decreases roughly exponentially.

b) Predictions overhead: The inference time of the used predictors is 2.8 microseconds per item on a single GPU without optimization, which ensures minimal impact on throughput. To evaluate memory overhead, we calculated the total number of parameters and corresponding memory requirements for each predictor model, which depends on the dataset. For the network flow predictor, the model has 167,521 parameters: 76,800 for IP address embeddings, 82,176 for encoding ports, and 8,545 for the frequency estimation network. With 32-bit float representations, this translates to around 0.67 MB of memory. The AOL dataset predictor is even more lightweight, with 7,680 parameters: 4,000 for character embeddings (RNN with LSTM cells), and 3,680 for the fully-connected layer, using approximately 0.03 MB of memory. Thus, the prediction overhead, which includes inference and memory, is therefore small. Note that inference overhead is expected to be less significant in the future [32] due to specialized hardware such as Google TPUs, hardware accelerators, and network compression [33], [34], [35], [36]. In terms of memory, there is a growing research field, TinyML [37], focused on creating tiny machine learning models for efficient on-device execution. It involves model compression techniques and high-performance system design for efficient ML. Here we present one example of predictors, which can be treated as black boxes without focusing on their internal functioning; our approach can therefore be used with any suitable and efficient learning scheme that yields a predictor, given the rapid advancements in machine learning research.

c) Robustness: Figure 4 shows the robustness of LSS that our theorems suggest using the web search dataset with a simulated predictor. Figure 4a evaluates the precision of top-k as function of the consumed space when $k = 10$, where the prediction for every item arrival is 1. Even in this extreme case, we observe only a small degradation in accuracy compared to SS. This degradation arises because we do not benefit from filtering since the filter is overloaded with items, causing items to be frequently replaced in the SS table. Additionally, the filter consumes some memory, so less memory is available for tracking items compared to SS. At the other extreme, Figure 4b shows the recall of finding heavy hitters when every item is predicted as a heavy hitter. In this scenario, the fixed entries in the counters could be filled with non-heavy hitter items, leaving fewer counters available for tracking the actual heavy hitters. This results in a decreased recall rate. However, once again, the decrease is small. Figure 4c illustrates the precision rate of the top-k task as a function of the prediction accuracy (p), as explained in the simulated predictor. When $p = 0$, it implies that all items are mispredicted, in which case SS yields a higher recall rate than LSS because the predictions provide no benefit. As p increases, LSS outperforms SS, and in the other extreme, when $p = 1$ (perfect prediction), LSS achieves around 50% improvement in precision compared to SS.

d) Accuracy vs. Fixed Counters: Figures 4d, 4e show the recall of finding top-k items and heavy hitters as a function of the number of fixed counters using the web search dataset with the learned model. As this number increases, the recall for identifying top-k items decreases because the fixed counters may be populated with heavy hitters that are not among the top-k items. Since these entries are fixed in a first-come manner, the top items may not be placed in the fixed counters. Allocating fixed counters for heavy hitters that are not top-k items results in fewer mutable counters for tracking top-k. However, as the number of fixed counters increases, the recall for detecting heavy hitters improves since having fixed counters dedicated to tracking heavy hitters aligns with this objective. Thus, for finding top-k items, we set the number of fixed entries to zero, while for finding heavy hitters, we allocated 10% of the counters as fixed counters.

e) Accuracy vs. Memory: We examine the accuracy of SS, LSS, and LSS+ (with $\tau = 0.5$) across three tasks: finding top-k items, identifying heavy hitters, and frequency estimation using web search, IP, and synthetic datasets. The accuracy is evaluated as a function of the memory used. Figure 5 shows the results for top-k and heavy hitter identification tasks. As expected, higher available memory results in improved precision and recall rates. For the top-k task, no fixed entries were used. LSS and LSS+ achieve better precision than SS when finding top-k items and better recall when identifying heavy hitters. LSS slightly outperforms LSS+, as one might



(a) All predictions are 1 (b) All predictions are heavy hitters (c) Predictions accuracy (d) Top-k vs. fixed counters (e) Heavy hitters vs. fixed counters

Fig. 4: (a-c) Robustness of LSS using web search dataset (a) precision of top-k ($k = 10$) with all predictions as 1 (b) recall of finding heavy hitters when all predictions are heavy hitters (c) precision of top-k vs. prediction accuracy p. (d-e) Impact of fixed counters on top-k ($k = 64$) and heavy hitters using web search dataset.

expect; here for both the filters we allocate allocate 10% of the memory. Figure 6 illustrates the RMSE of frequency estimation and shows the accuracy of each variant, LSS-LF and LSS-HH, separately. As expected, increasing memory consumption decreases RMSE. We observe that each variant improves the accuracy compared to SS, and the combined usage of techniques in LSS achieves higher accuracy in frequency estimation.

f) Accuracy vs. Filter Size: Figure 7a presents the impact of the filter ratio on the precision of top-k ($k = 64$) items using the IP dataset. In this experiment, we keep the threshold t fixed to default ($t = 4$) and vary only the filter ratio. Allocating less space to the filter does not affect LSS's correctness but it leads to a higher false positive rate for the filter. As a result, more items are included in the space-saving table, which affects accuracy and makes the approach more similar to falling back on the traditional space-saving algorithm.

g) Accuracy vs. Stream Length: Figure 7b shows the recall of finding heavy hitters using 2^{17} bit memory using web search dataset. At the beginning of the stream, LSS maintains an accurate result (high recall rate 1) by effectively filtering out low-frequency items. As the stream grows larger, medium items also accumulate, leading to a decrease in the recall rate for both methods.

h) Accuracy vs. t: Figure 7c displays the RMSE vs. t using a synthetic dataset with $\alpha = 1.3$ and $p = 0.9$. The RMSE decreases until a certain point and then increases. This behavior is due to the increasing false positive rate of the filter at larger values of t, which is related to the number of low-frequency items and the filter size. In general, with larger memory (here we used 2^{19} bit memory), the filter size can be increased proportionally, allowing it to handle more low-frequency items. However, the number of low-frequency items depends on the data distribution. If prior knowledge of the distribution is available, the filter size and t can be adjusted.

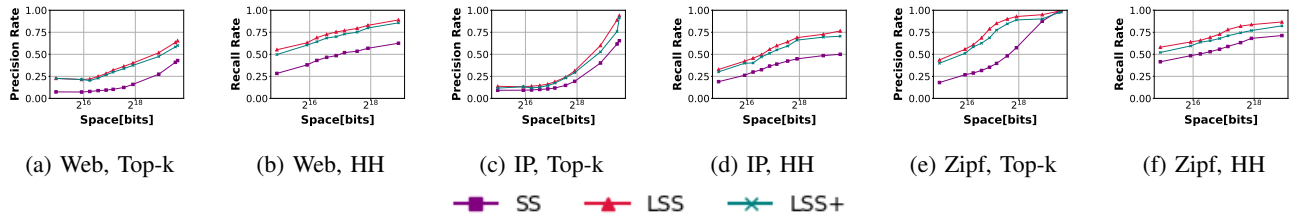
i) Accuracy vs. α: Using synthetic datasets with $p = 0.9$, Figure 7d shows the precision of top-k ($k = 124$) vs. α, the skewness parameter of a Zipfian distribution. As α increases, the distribution becomes more skewed, with a higher concentration of low-frequency items ("heavier tail"). LSS has improvements over SS until a certain point ($\alpha = 2$). After this point, LSS's precision starts to decrease due to the saturation of the filter with low-frequency items, resulting in higher false positive rates. When $\alpha \geq 2.2$, LSS has lower precision than SS since the filter becomes ineffective and fewer counters are allocated to the Space Saving table compared to SS.

j) Performance Comparison: Figure 6d examines the update performance of SS, LSS, LSS-HH, LSS-CBF and LSS+ algorithms using the IP dataset. We set $t = 1$ and use [38] and have not optimized further. A key consideration in comparing LSS to SS is the computational cost of inserting elements into the Bloom filter versus integrating them into the Space-Saving data structure. When an item is inserted or queried within a Bloom filter, additional hash computations take place. The performance of LSS degrades slightly due to the fact that insertion into the Bloom filter is less efficient than updating the Space-Saving data structure. Meanwhile, LSS+, configured with $\tau = 0.5$, noticeably outperforms both the aforementioned versions. This superior performance is due to its ability to minimize the number of insertions to the Bloom filter and to the SS. We skip query speed below since the discussed algorithms have the same query process. Figure 6e examines the update performance of LSS and LSS+ as a function of the parameter τ. (These experiments all use 32-bit floating point counters.) As τ increases, LSS+ saves more Bloom filter operations, resulting in improved update performance up to $\tau = 0.8$. Following this, LSS+ shows a slight drop in performance compared to LSS.

VIII. Related Works

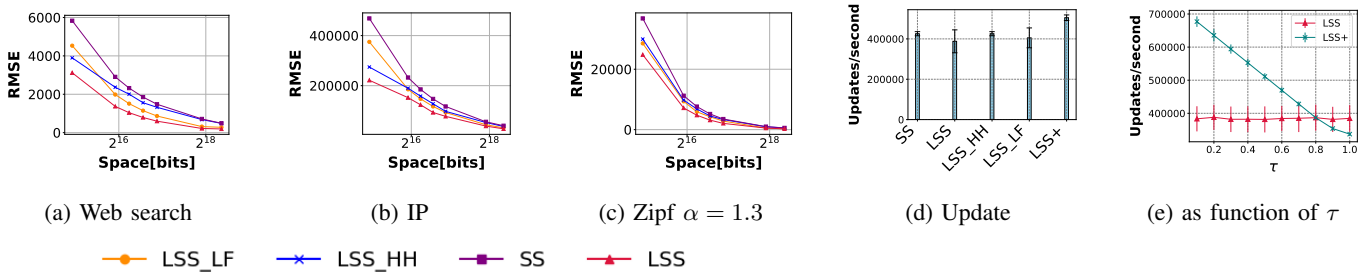
There are many algorithms proposed in the literature for frequency estimation, top-k, and identifying heavy hitters. See [18] for a survey.

Algorithms with predictions is, as we have stated, a rapidly growing area. The site [39] contains a collection of over a hundred papers on the topic. Related work on predictions for streaming algorithms [7], [40]–[43] introduced the concept of a heavy hitter predictor within the data stream which predicts whether it will be a heavy hitter or not. It was shown that such a predictor can reduce the estimation errors of classical hash-based algorithms, such as [14], [15], which approximately count item frequency based on a hashing process that maps



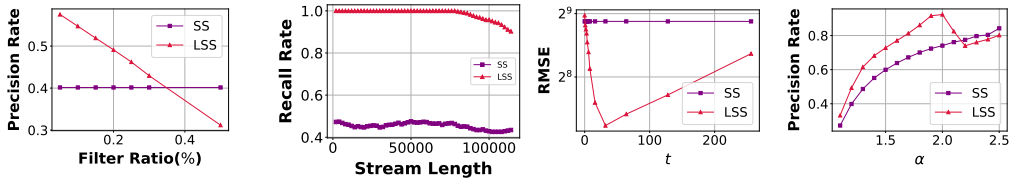
(a) Web, Top-k (b) Web, HH (c) IP, Top-k (d) IP, HH (e) Zipf, Top-k (f) Zipf, HH

Fig. 5: Precision and recall vs. memory of identifying top-k ($k = 64$) and heavy hitters. LSS+ configured with $\tau = 0.5$. We use web search, IP and Zipf ($\alpha = 1.3$) datasets.



(a) Web search (b) IP (c) Zipf $\alpha = 1.3$ (d) Update (e) as function of τ

Fig. 6: RMSE vs. memory of frequency estimation using web search, IP and Zipf ($\alpha = 1.3$) datasets.



(a) IP, Top-k (b) Web search, HH (c) Synthetic, Frequency (d) Synthetic, Top-k

Fig. 7: Impact of parameters (a) Precision vs. filter ratio of identifying top-k frequent item using IP dataset (b) recall rate for identifying heavy hitters in a web search dataset, focusing on the initial part of the stream (c) RMSE vs. t using synthetic dataset ($\alpha = 1.3$) (d) Precision vs. α of identifying top-k frequent item using synthetic dataset.

items to buckets. With the learning process, items that are predicted to be heavy hitters were not placed in the sketch but assigned their own counters, so they would be counted accurately. Intuitively, this not only gives these items accurate counts but also reduces the error of the items that use the sketch. Unfortunately, this approach does not directly translate to the competing counter-based as counters in this approach are not shared among items. Eliminating low cardinality hosts was also introduced by [44], which presents a two-phase filtering method specifically designed to identify high cardinality hosts in network traffic. Their technique, optimized for moderately high cardinalities, employs multi-stage sampling and multiple Bloom filters. In contrast, our approach uses a prediction-based method to identify and filter low-frequency items without relying on sampling and is designed as a general scheme adaptable to any stream distribution.

IX. Conclusion

Identifying heavy hitters and estimating the frequencies of flows are fundamental tasks in various network domains. Recent studies have used machine learning to enhance algorithms for approximate frequency estimation, but they focus solely on hashing-based methods, which may not be best for identifying heavy hitters. In this work, we have presented a novel learning-based approach for identifying heavy hitters, top k, and flow frequency estimation. We have applied this approach to the well-known Space Saving algorithm, which we have called Learned Space Saving (LSS). Our approach is designed to be resilient against prediction errors.We demonstrated our design's benefits analytically and empirically. Experiments on real-world datasets show that LSS achieves higher recall, precision, and improved RMSE compared to the traditional Space Saving algorithm. Code is available online [45].

Acknowledgments

We thank our Sheperd Jun Xu and and the anonymous reviewers for helping improve our paper.

REFERENCES

[1] Y. Azar, S. Leonardi, and N. Touitou, "Distortion-oblivious algorithms for minimizing flow time," in *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, J. S. Naor and N. Buchbinder, Eds. SIAM, 2022, pp. 252–274. [Online]. Available: https://doi.org/10.1137/1.9781611977073.13

[2] ——, "Flow time scheduling with uncertain processing time," in *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, S. Khuller and V. V. Williams, Eds. ACM, 2021, pp. 1070–1080. [Online]. Available: https://doi.org/10.1145/3406325.3451023

[3] M. Mitzenmacher, "Queues with small advice," in *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2021, Virtual Conference, July 19-21, 2021*, M. Bender, J. Gilbert, B. Hendrickson, and B. D. Sullivan, Eds. SIAM, 2021, pp. 1–12. [Online]. Available: https://doi.org/10.1137/1.9781611976830.1

[4] Z. Scully, I. Grosof, and M. Mitzenmacher, "Uniform bounds for scheduling with job size estimates," in *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA*, ser. LIPIcs, M. Braverman, Ed., vol. 215. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 114:1–114:30. [Online]. Available: https://doi.org/10.4230/LIPIcs.ITCS.2022.114

[5] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *Journal of the ACM (JACM)*, vol. 68, no. 4, pp. 1–25, 2021.

[6] D. Rohatgi, "Near-optimal bounds for online caching with machine learned advice," in *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, S. Chawla, Ed. SIAM, 2020, pp. 1834–1845. [Online]. Available: https://doi.org/10.1137/1.9781611975994.112

[7] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms." in *International Conference on Learning Representations*, 2019.

[8] G. Dittmann and A. Herkersdorf, "Network processor load balancing for high-speed links," in *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, vol. 735. Citeseer, 2002.

[9] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers," in *2010 18th ieee symposium on high performance interconnects*. IEEE, 2010, pp. 58–65.

[10] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network intrusion detection," *IEEE network*, vol. 8, no. 3, pp. 26–41, 1994.

[11] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.

[12] G. Einziger, R. Friedman, and B. Manes, "Tinylfu: A highly efficient cache admission policy," *ACM Transactions on Storage (ToS)*, vol. 13, no. 4, pp. 1–31, 2017.

[13] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Accurate and efficient sla compliance monitoring," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007, pp. 109–120.

[14] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[15] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[16] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou, "Dynamically maintaining frequent items over a data stream," in *Proceedings of the twelfth international conference on Information and knowledge management*, 2003, pp. 287–294.

[17] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer, 2005, pp. 398–412.

[18] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[19] ——, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, pp. 3–20, 2010.

[20] P. Hick, "CAIDA Anonymized Internet Trace, equinix-chicago," 2016.

[21] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[22] M. Mitzenmacher and S. Vassilvitskii, "Algorithms with predictions," *Communications of the ACM*, vol. 65, no. 7, pp. 33–35, 2022.

[23] F. Zhao, D. Agrawal, A. E. Abbadi, and A. Metwally, "Spacesaving+-: An optimal algorithm for frequency estimation and frequent items in the bounded deletion model," *arXiv preprint arXiv:2112.03462*, 2021.

[24] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 286–296.

[25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[26] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[27] P. C. Dillinger and S. Walzer, "Ribbon filter: practically smaller than bloom and xor," *arXiv preprint arXiv:2103.02515*, 2021.

[28] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[29] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[30] "Web search query log," https://jeffhuang.com/search_query_logs/.

[31] D. M. Powers, "Applications and explanations of zipf's law," in *New methods in language processing and computational natural language learning*, 1998.

[32] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[33] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[34] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[35] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[36] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[37] L. Dutta and S. Bharali, "Tinyml meets iot: A comprehensive survey," *Internet of Things*, vol. 16, p. 100461, 2021.

[38] Michael Axiak, "pybloomfiltermmap3," https://github.com/prashnts/pybloomfiltermmap3.

[39] https://algorithms-with-predictions.github.io/, "Algorithms with predictions github repository."

[40] R. Gupta and T. Roughgarden, "A pac approach to application-specific algorithm selection," in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, 2016, pp. 123–134.

[41] T. Dick, M. Li, V. K. Pillutla, C. White, N. Balcan, and A. Smola, "Data driven resource allocation for distributed learning," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 662–671.

[42] M.-F. Balcan, T. Dick, and E. Vitercik, "Dispersion for data-driven algorithm design, online learning, and private optimization," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018, pp. 603–614.

[43] T. Jiang, Y. Li, H. Lin, Y. Ruan, and D. P. Woodruff, "Learning-augmented data stream algorithms," in *International Conference on Learning Representations*, 2019.

[44] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang, "Identifying high cardinality internet hosts," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 810–818.

[45] “Open source code,” https://anonymous.4open.science/r/LearnedSS-2B31.