LotterySampling: A Randomized Algorithm for the Heavy Hitters and Top-k Problems in Data Streams*

Conrado Martínez[1][0000−0003−1302−9067] and Gonzalo Solera-Pardo[1,2][0000−0001−5668−1210]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain conrado@cs.upc.edu, gonzalo.solera@estudiantat.upc.edu
[2] Google Research Labs, Zürich, Switzerland gonsp@google.com

Abstract. We propose a new randomized count-based algorithm to solve the Heavy Hitters and Top-k problems in data streams. This algorithm, called LotterySampling, uses the intuitive concept of "lottery tickets", to decide which elements to sample. We prove that LotterySampling is inside the (δ, ϵ)-deficient framework for the Heavy Hitters problem and that it has a similar performance to the well known StickySampling algorithm, although they are very different in nature. More importantly, we define a similar (δ, ϵ)-deficient framework for the harder Top-k problem and we prove that LotterySampling is inside it. Hence, LotterySampling can be used, without any previous assumption of the data distribution, as a probabilistic approximation scheme to find the k most frequent elements and to approximate the frequencies of the reported elements by a factor of $1 - \epsilon$. To the best of our knowledge, this is the first algorithm that gives theoretical guarantees for the Top-k problem for unknown and arbitrary streams, which is the most important contribution of this paper. Its memory usage is adaptive to the distribution of the stream, and it will increase or decrease depending on whether the stream becomes less or more skewed. More precisely, the sample size depends, at any given moment, on the unknown k^{th} highest frequency but it is independent of the length of the stream and of the number of distinct elements. The user just needs to provide two parameters that determine the quality of the answers, independently of the stream. We compare LotterySampling with other existing probabilistic and deterministic algorithms showing its strengths and weaknesses.

1 Introduction

The Heavy Hitters and Top-k problems are two heavily studied problems in data stream mining. Given a long data stream, consisting in a sequence of elements

* This work has been supported by funds from the MOTION Project (Project PID2020-112581GB-C21) of the Spanish Ministry of Science & Innovation MCIN/AEI/10.13039/501100011033, while the second author was a postgraduate student at Universitat Politècnica de Catalunya.

(with repetitions), we are interested in detecting what the most frequent elements are, and to estimate their frequencies.

This might seem a simple problem, easily solved by keeping a frequency counter for each distinct element that appears in the stream. But it is quite harder than that, since the number of distinct elements might be intractable in massive data streams. For example, consider a router that wants to identify the most frequent IP addresses that pass through it, in order to prevent a DDoS attack. The number of distinct IP addresses is too big to store a counter for each address. Instead, we will have to content ourselves by approximating the most frequent elements (heavy hitters) and their frequencies.

This is achieved by keeping a subset of elements in a sample of some tractable size, with many possible strategies about how to update this sample and the estimated frequencies of the elements in it. For a good introduction to the problem, several well-known algorithms and their analysis, we refer the reader to [7,8].

The rest of this paper is organized as follows. In Section 2 we introduce some notation and formalize the two problems that we will consider here: Heavy Hitters and Top-k. We also recall the standard definitions of ϵ-deficient and (δ, ϵ)-deficient algorithms for Heavy Hitters, and introduce a similar, but novel, definitions of ϵ-deficiency and (δ, ϵ)-deficiency for Top-k algorithms. Section 3 is devoted to present the common concepts and intuition behind our two novel algorithms first, then we present the two algorithms themselves: LotterySampling for Heavy Hitters (subsection 3.1) and LotterySampling for Top-k (subsection 3.2) and prove that they are both (δ, ϵ)-deficient—in their respective contexts. While there are several ϵ- and (δ, ϵ)-deficient algorithms for Heavy Hitters, our algorithm for Top-k is the first algorithm providing guarantees for the Top-k problem when no prior knowledge of the stream is available, and hence beating the state-of-the-art (to the best of our knowledge) in this specific setup. After that, Section 4 reports some experimental results and compares our algorithm with two well-known alternatives, namely, StickySampling [6] and SpaceSaving [7]. We conclude with some final remarks in Section 5.

2 Problem formulation

Consider a (long) data stream $\mathcal{Z} = (z_1, \ldots, z_N)$, where each z_i is an occurrence of an element x_j drawn from some domain $\mathcal{X} = \{x_1, \ldots, x_n\}$ of large cardinality $n \leq N$.

We will use f_x to denote the number of occurrences (absolute frequency) of x in $\mathcal{Z}$. We will assume, w.l.o.g., that we index the elements in $\mathcal{X}$ in non-increasing order of frequency, thus $f_{x_1} \geq f_{x_2} \geq \cdots \geq f_{x_{n-1}} \geq f_{x_n} > 0$. We will use $p_x = f_x/N$ to denote the relative frequency of x.

Because of the restrictions of the data stream model, algorithms for the Heavy Hitters or the Top-k problems cannot store all the n distinct elements x nor their frequencies f_x. Instead, count-based algorithms maintain estimated frequencies f'_x of the elements stored (or sampled) in a sample $S \subseteq \mathcal{X}$, which aims to contain the most frequent elements.

In the rest of this paper, we will estimate the frequency of x using the observed frequency of x, that is, f'_x will be the number of occurrences of x since the last time it was inserted into S. Similarly as before, we will refer to the element x with j^{th} highest estimated frequency f'_x by $\hat{x}_j$, assuming that $f'_{\hat{x}_j} = 0$ if $\hat{x}_j \notin S$ and solving ties arbitrarily. Therefore, x_j and $\hat{x}_j$ are not necessary the same element.

The two problems (or type of queries) that we want to study here are the Heavy Hitters and the Top-k problems, which will be discussed in the following two subsections.

2.1 Heavy Hitters

Given $\mathcal{Z}$ and a value $0 < \phi < 1$, we want to find all the elements x that appear in $\mathcal{Z}$ with relative frequency $p_x \geq \phi$. Those elements are called *heavy hitters*. In other words, we want to obtain $\{x_1, \ldots, x_{k^*}\}$, where k^* is the largest value k such that $p_{x_k} \geq \phi$. The value k^* is the number of heavy hitters.

A popular yardstick to compare heavy hitters algorithms, often used in the literature, is the ϵ-*deficient* framework [6,7,8,3,1]. We say that an algorithm is ϵ-deficient if the following properties are satisfied:

1. All the heavy hitters (elements x with $f_x \geq \phi N$) are reported. i.e., there are no false negatives.
2. No element x with $f_x < (\phi - \epsilon)N$ is reported.
3. For every reported element x, its estimated frequency f'_x satisfies: $f_x - \epsilon N \leq f'_x \leq f_x$, or equivalently, $p_x - \epsilon \leq p'_x \leq p_x$.

A probabilistic algorithm is (δ, ϵ)-deficient if the probability of not satisfying all the previous properties is at most δ. Here the probability is defined with respect the algorithm's random choices, not with respect an input distribution—our analysis assumes the worst-case input distribution.

2.2 Top-k

Given $\mathcal{Z}$ and a value $k \leq n$, we want to find $\{x_1, \ldots, x_k\}$ (solving ties arbitrarily when $f_{x_k} = f_{x_{k+1}}$). Sometimes we will call these top-k elements heavy hitters, a common abuse in the literature, if no confusion arises. We propose another (δ, ϵ)-deficient framework very similar to the one used in the Heavy Hitters problem:

1. All the heavy hitters (elements $x_1, \ldots, x_k$) are reported. i.e., there are no false negatives.
2. No element x with $f_x < (1 - \epsilon) f_{x_k}$ is reported.
3. For every reported element x, its estimated frequency f'_x satisfies: $(1-\epsilon) f_x \leq f'_x \leq f_x$, or equivalently, $(1 - \epsilon) p_x \leq p'_x \leq p_x$.

Although the differences between Top-k and Heavy Hitters are subtle, they should not be overlooked, since they make the Top-k problem significantly harder

than the Heavy Hitters problem. The main reason is that for a given k, the frequencies $f_{x_1}, \ldots, f_{x_k}$ can be arbitrarily small and/or arbitrarily close to the remaining frequencies (e.g. $f_{x_k} = f_{x_{k+1}} + 1$) making more difficult to give accuracy guarantees and to choose an appropriate sample size $|S|$, since f_{x_k} (or more precisely, p_{x_k}) is unknown. In fact, to solve Top-k deterministically (exactly) implies the use of a linear number of counters in n [5], which is obviously unacceptable.

3 The algorithms

The algorithms that we propose are based on a new and original idea that consists in the concept of *lottery tickets* and *lottery tokens*. A lottery token (or token) is an independent uniform random number between 0 and 1. Basically, we will generate one of such tokens t_i for each occurrence z_i. And each element x will have its own lottery ticket (or ticket) T_x, which will be the highest token obtained among all its occurrences so far.

Hence, we can consider the tokens t_i and the tickets T_x as random variables, and their probability distributions are $\mathbb{P}[t_i \leq y] = y$, and $\mathbb{P}[T_x \leq y] = y^{f_x}$, respectively and thus the expectation of T_x is:

$$\mathbb{E}[T_x] = \frac{f_x}{f_x + 1}.$$

Note that, for any given x_a and x_b, $\mathbb{E}[T_{x_a}] \geq \mathbb{E}[T_{x_b}]$ if and only if $f_{x_a} \geq f_{x_b}$. Hence, $\mathbb{E}[T_{x_1}] \geq \mathbb{E}[T_{x_2}] \geq \ldots \geq \mathbb{E}[T_{x_n}]$. This is the powerful idea that we try to exploit in our algorithms: Consider a naïve algorithm that, although it does not know the frequencies f_x, it easily keeps track of the k-highest tickets T_x, expecting to find the top-k heavy hitters with only $\mathcal{O}(k)$ counters and to solve the problem exactly. The algorithm would be using the ticket of an element to approximate its frequency. Obviously, this approach wouldn't work well since $\mathrm{Var}[T_x] > 0$. More precisely, $\mathbb{P}[T_i > T_j] = \frac{f_{x_i}}{f_{x_i} + f_{x_j}}$, which means that the probability of having two elements in inverted order is high when their frequencies are similar. However, this naïve algorithm is useful to give an intuition of the more elaborated algorithms that we propose in this paper.

The reason behind using the name "lottery ticket" is that we can define an analogy with a regular lottery: In a regular lottery there are some winners between the participants, and they are randomly chosen by selecting a subset of lottery numbers. The more lottery numbers a participant has, the more chances it will have of winning. Similarly in the naïve algorithm, the more frequent an element is, the more lottery tokens it will get, so it will stand a better chance of getting a high lottery ticket. If its lottery ticket is among the k-highest lottery tickets, it will be reported by the previous algorithm, or following the analogy, be one of the winners of the lottery.

The algorithms below will deal with the variance of the tickets T_x and they will provide probabilistic guarantees of their accuracy and efficiency. Note that

the tokens and tickets will be represented with a fixed (and arbitrary) number of bits, and the more bits used, the closer the result will be to the theoretical analysis. However, 64 bits will be more than enough in any practical scenario.

3.1 LotterySampling for Heavy Hitters

LotterySampling for the Heavy Hitters problem will generate a token t_i for each occurrence z_i and it will keep in the sample S all the elements x with a ticket T_x above a threshold $\tau < 1$. This threshold evolves as the algorithm processes the stream and it will always increase (faster at the beginning, more slowly as more occurrences from the data stream have been processed). The underlying idea is to use the expected ticket of an imaginary element x with frequency $f_x = \epsilon N$ as the threshold, which would be equal to $\mathbb{E}[T_x] = \frac{\epsilon N}{\epsilon N+1}$. To simplify the calculations, we will use a slightly modified threshold instead: $\frac{\epsilon N-1}{\epsilon N}$. This way, since the elements with frequency above ϵN have an expected ticket above the threshold, we can expect having them sampled in S. Due to the variance of the tickets, we will need to scale down this threshold using some value r, so we can probabilistically guarantee that the heavy hitters are inserted before they overpass the frequency ϵN and that they will stay in the sample since then. Hence, the threshold will finally be: $\tau = \frac{\epsilon N/r-1}{\epsilon N/r} = 1 - \frac{r}{\epsilon N}$. LotterySampling for the Heavy Hitters problem is formally defined in Algorithm 1.

Algorithm 1: LotterySampling(ϕ, ϵ, δ)

```
S := ∅; r := ln(φ^{-1}δ^{-1}); N := 0
for z_i ∈ 𝒵 do
    N := N + 1; t_i := ticket(z_i)
    τ := 1 − r/(εN)
    if z_i ∈ S then
        f'_{z_i} := f'_{z_i} + 1; T_{z_i} := max(T_{z_i}, t_i)
    else if t_i ≥ τ then
        S.insert(z_i); f'_{z_i} := 1; T_{z_i} := t_i
    S.delete({x ∈ S | T_x < τ})
/* The algorithm can answer a Heavy Hitters query at any moment by
   reporting all the elements x in S such that: f'_x ≥ (φ − ε)N.   */
```

For the following analysis, let $\tau(i)$ be the value of the threshold τ when processing z_i. Hence, $\tau(N)$ will refer to the current threshold.

Lemma 1. *Let z_i with $i \leq N$ be an occurrence of the element x such that $t_i \geq \tau(N)$. Then, x will be in S at least from iteration i until iteration N.*

Proof. Since $\tau(i+1) \geq \tau(i)$ for all i (τ never decreases), then $t_i \geq \tau(N) \geq \tau(N-1) \geq \ldots \geq \tau(i)$. In particular $t_i \geq \tau(i)$, so x will enter S if it wasn't already

being sampled in the i^{th} iteration. Note that if an element x is evicted from S, its ticket T_x is lower than the threshold at that moment. Since $T_x \geq t_i \geq \tau(N)$, x cannot be evicted before the N^{th} iteration.

Lemma 2. *Let z_i with $i \leq N$ be an occurrence of the element x among its first μ occurrences such that $t_i \geq \tau(N)$. Then, its observed frequency f'_x will be greater or equal than $f_x - \mu$.*

Proof. From Lemma 1 we know that x will at least be in the sample between the i^{th} and the N^{th} iterations. So all of its occurrences between the i^{th} and the N^{th} will be accounted for in its observed frequency f'_x counter (because the element will be in the sample during that time). Hence, all of its occurrences that might not be accounted for in its estimated frequency f'_x are the previous ones, of which there are at most μ, by hypothesis.

Lemma 3. *For any element x, $f'_x \leq f_x$.*

Proof. f'_x is always initialized with 1, and every time it is increased is due to an actual occurrence of x. Hence, f'_x never overestimates f_x.

Theorem 1. *LotterySampling for Heavy Hitters is a (δ, ϵ)-deficient algorithm.*

Proof. For a sampled element x with $f_x < (\phi - \epsilon)N$, from Lemma 3 we know that $f'_x < (\phi - \epsilon)N$, so x won't be reported by the algorithm, satisfying property 2 of an ϵ-deficient algorithm.

If an element x with frequency f_x such that $(\phi - \epsilon)N \leq f_x < \phi N$ is reported (false positive), it is because its estimated frequency satisfies $(\phi - \epsilon)N \leq f'_x$. Applying Lemma 3 again, we get that $f_x - \epsilon N < f'_x \leq f_x$. Hence, the property 3 of an ϵ-deficient algorithm is not violated when reporting false positives.

For every heavy hitter x_j, let the indicator random variable Y_j be 1 if the heavy hitter x_j obtains a token greater or equal than $\tau(N)$ among its first ϵN occurrences, and let Y_j be 0 otherwise. If $Y_j = 1$, from Lemma 1 we know that x_j will be in S, and from Lemma 2 we know that $f'_{x_j} \geq f_{x_j} - \epsilon N$. Since x_j is a heavy hitter ($f_{x_j} \geq \phi N$), then $f'_{x_j} \geq (\phi - \epsilon)N$ which means that it will be reported, satisfying property 1 of an ϵ-deficient algorithm. Property 3 would be satisfied as well, as can be seen by applying Lemma 3 again.

Hence, if $Y_1 = 1, \ldots, Y_{k^*} = 1$, then all the properties of an ϵ-deficient algorithm would be satisfied. Note that this is a sufficient, but not necessary condition. For a heavy hitter x_j, the probability of not obtaining a token larger or equal than $\tau(N)$ among its first ϵN occurences (and thus, x_j not being sampled and/or reported as a heavy hitter) is $\mathbb{P}[Y_j = 0] = (1 - \frac{r}{\epsilon N})^{\epsilon N}$. Since $k^* \leq \lfloor 1/\phi \rfloor$, we can apply the union bound to obtain:

$$\mathbb{P}[\text{failure}] \leq \mathbb{P}\left[\bigvee_{j=1}^{k^*} Y_j = 0\right] \leq \frac{1}{\phi}\left(1 - \frac{r}{\epsilon N}\right)^{\epsilon N} \leq \phi^{-1} e^{-r}$$

Thus, $\mathbb{P}[\text{failure}] \leq \delta$ if $\phi^{-1} e^{-r} \leq \delta$, that is, if $r \geq \ln(\phi^{-1}\delta^{-1})$.

Theorem 2. *The expected sample size is* $\mathcal{O}(\frac{1}{\epsilon}\ln(\phi^{-1}\delta^{-1}))$.

Proof. When $N < \frac{r}{\epsilon}$ then $\tau(N) < 0$ which implies that all the elements are inserted into S since $t_i \geq 0$. When $N \geq \frac{r}{\epsilon}$, the worst case of memory consumption —measured by the number of sampled elements— happens when every element has a single occurrence in the stream. In that case, the number of elements with ticket greater or equal than the threshold $\tau(N)$ follows a binomial distribution with success probability $\frac{r}{\epsilon N}$ and length N. Hence:

$$\mathbb{E}[|S|] \leq \frac{r}{\epsilon N}N = \frac{1}{\epsilon}\ln(\phi^{-1}\delta^{-1}).$$

Theorem 3. *The execution time per processed occurrence in the data stream is* $\mathcal{O}(\log(|S|))$.

Proof. Keeping the elements of S in order of observed frequency is possible in $\mathcal{O}(1)$ time by using the data structure proposed in [7], allowing to answer Heavy Hitters queries in linear time respect to the number of elements with $f'_x > (\phi - \epsilon)N$, which is optimal. The main complication lies in the purge of S, where we delete all the elements with ticket below τ. In the worst case, all the elements would need to be removed at the same time (very unlikely but possible). In such case, with a straightforward implementation, the cost would be $\mathcal{O}(|S|)$. To solve this, we propose to purge S lazily: Only when an element x that wasn't in the sample needs to be inserted (because $T_x \geq \tau$), we purge $\mathcal{O}(1)$ elements with ticket below τ. This way, every time $|S|$ increases, it is because all the elements in S have a ticket greater or equal than τ. If there was at least one element with ticket below τ, the sample size wouldn't increase. Therefore, the previous space bound from Theorem 2 is still valid.

Besides that data structure we will also need a min-heap of the tickets, which yields a total time cost of $\mathcal{O}(\log|S|)$ per processed occurrence in the worst case— to update the ticket if the processed occurrence z_N was already in the sample or to insert the ticket t_N if the processed occurrence z_N was not in the sample and it has to be added to the sample. However, note that for most of the occurrences no action will need to be taken since the lottery token won't be greater than the threshold τ.

3.2 LotterySampling for Top-k

LotterySampling for the Top-k problem is very similar to the Heavy Hitters version. The main difference is in how the threshold is defined: $\tau = 1 - \frac{r}{(1-\sqrt{1-\epsilon})f'_{\hat{x}_k}}$, which uses the k^{th} highest observed frequency $f'_{\hat{x}_k}$. Also, an element is only evicted from S if its ticket is below τ and it isn't one of the k elements with highest observed frequency $\{\hat{x}_1, \ldots, \hat{x}_k\}$. LotterySampling for the Top-k problem is formally defined in Algorithm 2.

Lemma 4. *The threshold* τ *never decreases. i.e.,* $\tau(i) \leq \tau(i+1)$ *for all* i.

Algorithm 2: LotterySampling(k, ϵ, δ)

$S := \emptyset$; $r := \ln(k/\delta)$
for $z_i \in \mathcal{Z}$ **do**
  $N := N + 1$; $t_i := \text{ticket}(z_i)$
  $\tau := 1 - \frac{r}{(1-\sqrt{1-\epsilon})f'_{\hat{x}_k}}$ /* Use $\tau = 0$ if the size of S is $< k$ */
  if $z_i \in S$ **then**
    $f'_{z_i} := f'_{z_i} + 1$; $T_{z_i} := \max(T_{z_i}, t_i)$
  else if $t_i \geq \tau$ **then**
    S.insert(z_i); $f'_{z_i} := 1$; $T_{z_i} := t_i$
  S.delete$(\{\hat{x}_j \in S \mid j > k \wedge T_{\hat{x}_j} < \tau\})$
/* The algorithm can answer a Top-k query at any moment by reporting all the elements x in S such that: $f'_x \geq \sqrt{1-\epsilon} f'_{\hat{x}_k}$. */

Proof. By construction of the algorithm, the current k elements with highest observed frequencies $\{\hat{x}_1, \ldots, \hat{x}_k\}$ are never removed from S. Hence $f'_{\hat{x}_k}$ never decreases, and neither does τ.

Lemma 5. *Lemmas 1, 2 and 3 also hold for this version of LotterySampling.*

Proof. The proofs are the same, using the fact that τ never decreases from Lemma 4.

Lemma 6. *The k^{th} highest observed frequency is a lower bound of the k^{th} highest frequency. i.e., $f'_{\hat{x}_k} \leq f_{x_k}$.*

Proof. Assume that $f'_{\hat{x}_k} > f_{x_k}$. Since $f'_{\hat{x}_1} \geq \ldots \geq f'_{\hat{x}_k} > f_{x_k}$, then all the distinct elements $\{\hat{x}_1, \ldots, \hat{x}_k\}$ must have appeared more than f_{x_k} many times, so the k^{th} highest frequency should be greater than f_{x_k} which is a contradiction.

Similarly as in the Heavy Hitters version, let $Y_j = 1$ for every heavy hitter x_j if it obtains a token greater or equal than $\tau(N)$ among its first $(1 - \sqrt{1-\epsilon})f_{x_k}$ occurrences, and let $Y_j = 0$ otherwise.

Lemma 7. *If $Y_1 = 1, \ldots, Y_k = 1$, then for every heavy hitter x, its observed frequency is $f'_x \geq \sqrt{1-\epsilon} f_x$.*

Proof. For every heavy hitter x, applying Lemma 2 we get that $f'_x \geq f_x - (1 - \sqrt{1-\epsilon})f_{x_k}$. Knowing that $f_x \geq f_{x_k}$ since x is a heavy hitter we obtain $f'_x \geq \sqrt{1-\epsilon} f_x$.

Lemma 8. *If $Y_1 = 1, \ldots, Y_k = 1$, then $\sqrt{1-\epsilon} f_{x_k} \leq f'_{\hat{x}_k} \leq f_{x_k}$.*

Proof. Since there are k heavy hitters, and from Lemma 7 all of them have $f'_x \geq \sqrt{1-\epsilon} f_x \geq \sqrt{1-\epsilon} f_{x_k}$, then $f'_{\hat{x}_k}$ cannot be smaller than $\sqrt{1-\epsilon} f_{x_k}$ since by definition $f'_{\hat{x}_k}$ is the k^{th} highest observed frequency. i.e., $\sqrt{1-\epsilon} f_{x_k} \leq f'_{\hat{x}_k}$. The other side of the inequality comes from Lemma 6.

Theorem 4. *LotterySampling for Top-k is a (δ, ϵ)-deficient algorithm.*

Proof. When $Y_1 = 1, \ldots, Y_k = 1$, all the properties of an ϵ-deficient algorithm are satisfied. First, note that by Lemma 1 every heavy hitter x would be sampled, and by Lemmas 7 and 6, x would be reported since $f'_x \geq \sqrt{1-\epsilon} f_x \geq \sqrt{1-\epsilon} f_{x_k} \geq \sqrt{1-\epsilon} f'_{\hat{x}_k}$. By lemmas 7 and 3, its estimated frequency satisfies $(1-\epsilon) f_x \leq \sqrt{1-\epsilon} f_x \leq f'_x \leq f_x$. Second, note that by Lemmas 3 and 8, any reported element x that isn't a heavy hitter satisfies: $f_x \geq f'_x \geq \sqrt{1-\epsilon} f_{\hat{x}_k} \geq (1-\epsilon) f_{x_k} \geq (1-\epsilon) f_x$.

Finally, using that $f_{\hat{x}_k} \leq f_{x_k}$ from Lemma 6 we obtain that for any heavy hitter x_j:

$$\begin{aligned}\mathbb{P}[Y_j = 0] &= \left(1 - \frac{r}{(1-\sqrt{1-\epsilon}) f_{\hat{x}_k}}\right)^{(1-\sqrt{1-\epsilon}) f_{x_k}} \\ &\leq \left(1 - \frac{r}{(1-\sqrt{1-\epsilon}) f_{x_k}}\right)^{(1-\sqrt{1-\epsilon}) f_{x_k}} \leq e^{-r}.\end{aligned}$$

Using the union bound we have $\mathbb{P}[\text{failure}] \leq \mathbb{P}\left[\bigvee_{j=1}^{k} Y_j = 0\right] \leq k e^{-r}$. Thus, $\mathbb{P}[\text{failure}] \leq \delta$ if $k e^{-r} \leq \delta$, that is, if $r \geq \ln(k/\delta)$.

Theorem 5. *If $Y_1 = \cdots, Y_k = 1$, then the expected size of the sample is $\mathcal{O}\left(\frac{\log(k/\delta)}{\sqrt{1-\epsilon}(1-\sqrt{1-\epsilon}) p_{x_k}}\right)$, which is only dependent on p_{x_k} (and thus independent of N if p_{x_k} is independent of N).*

Proof. The proof is very similar to the one for Theorem 2 for the Heavy Hitters version. We need to use that $f'_{\hat{x}_k} \geq \sqrt{1-\epsilon} f_{x_k}$ from Lemma 8, obtaining

$$\mathbb{E}[|S|] \leq \frac{r\,N}{(1-\sqrt{1-\epsilon}) f'_{\hat{x}_k}} \leq \frac{r\,N}{\sqrt{1-\epsilon}(1-\sqrt{1-\epsilon}) f_{x_k}} = \frac{\log(k/\delta)}{\sqrt{1-\epsilon}(1-\sqrt{1-\epsilon}) p_{x_k}}.$$

Theorem 6. *Each data stream occurrence is processed in time $\mathcal{O}(\log(|S|))$.*

Proof. The proof is analogous to the one for Theorem 3.

4 Comparison with previous work

4.1 Heavy Hitters

A similar and well-known probabilistic algorithm for the Heavy Hitters problem is StickySampling [6]. However, LotterySampling has important differences: (1) StickySampling uses a counting sample [4], and LotterySampling does not; (2) LotterySampling uses half the counters than StickySampling does, although LotterySampling needs to also keep the lottery ticket of every sampled element; (3) StickySampling has amortized $\mathcal{O}(1)$ execution time per stream occurrence (but $\mathcal{O}(|S|)$ in the worst case), while LotterySampling has $\mathcal{O}(\log(|S|))$ in the worst case; and (4) LotterySampling can answer Heavy Hitters and Top-k queries in

linear time with respect to the number of reported elements, which is optimal. StickySampling cannot keep the elements in order without incurring in a higher cost per stream occurrence.

In general, although LotterySampling is a simple and original probabilistic algorithm, it isn't competitive for the Heavy Hitters problem, since state-of-the-art deterministic count-based algorithms like SpaceSaving [7] have better precision and use less memory. This is well illustrated by the experiments that we have conducted and explained in Appendix A.1 (see Fig. 1).

4.2 Top-k

As already mentioned in section 2, the Top-k problem is much harder than the Heavy Hitters problem, since the frequency of the k most frequent elements is unknown and it can be arbitrarily small and similar to the rest of elements. Solving exactly the Top-k problem is intractable since $\Omega(N)$ counters are needed.

It follows that any algorithm that tries to answer Top-k queries cannot have a fixed and constant memory usage (sample size), since the distribution of the stream might radically change over time. LotterySampling adapts to such changes of the distribution and its sample size will increase or decrease accordingly. This way, we can give (probabilistic) guarantees about the performance of LotterySampling for unknown streams, while offering an upper bound of sample size independent of N.

Algorithms like SpaceSaving or CountSketch [2] also tackle the Top-k problem. However, their memory usage is fixed beforehand and constant through time, so they bypass the previously mentioned theoretical impossibility by assuming an arbitrary stream with known N and/or p_{x_k}, or a Zipfian stream with known parameters. This makes the Top-k problem artificially and noticeably easier, since p_{x_k} becomes known. Under that setup, the Top-k problem would indeed become very similar to the Heavy Hitters problem (since $\phi = p_{x_k}$). However, for arbitrary and unknown streams, where neither the length N nor p_{x_k} are known beforehand, no previously existing algorithm is able to give guarantees.

This means that it does not make much sense to experimentally compare LotterySampling with other algorithms that do not solve the Top-k problem for unknown and arbitrary streams, since algorithms like SpaceSaving need to fix their sample size depending on the unknown p_{x_k}. However, an experimental comparison between LotterySampling and SpaceSaving is presented Appendix A.2 (see Fig. 2), where we used an artificially constructed stream named ZipfNoiseZipf in which different "sections" of the stream exhibit markedly different properties; a detailed description of the model to generate the input stream is given there.

The experiments show that SpaceSaving does not adapt to the change of distribution (since its sample size is fixed as an initial parameter), and its precision and recall can be significantly affected in the streams that show very different "regimes". However, LotterySampling automatically adapts to the increasingly difficult stream using more memory as needed, and frees it up once the stream becomes skewed again, all while maintaining a theoretically guaranteed precision and recall.

In other words, the user of LotterySampling just needs to choose values for its parameters depending on the desired quality of its answers, and independently from the (unknown) stream's distribution. In the case of SpaceSaving, the user cannot choose the parameters (sample size) that guarantee a specific quality of the answers, since they depend on the stream distribution.

On the other hand, the experiments also show SpaceSaving would perform better than LotterySampling if using the same memory. Hence, one might think that using SpaceSaving with all the available memory will always be better than using LotterySampling. However, consider a case where we want to find the respective Top-k most frequent elements from several different streams in the same machine, which is limited by some fixed memory. How much memory should we assign to each instance of SpaceSaving? We do not know, since the streams' distributions are unknown. Equally splitting all the available memory might be sub-optimal since there might be "easy" and "difficult" streams. Instead, with LotterySampling we just choose the parameters depending on the desired quality and each instance will consume the memory that it needs, depending on the unknown (and variable) stream's skewness.

Furthermore, there is no clear strategy about how many sampled elements SpaceSaving should return when answering a Top-k query when trying to guarantee that all the k most frequent elements are retrieved, and the default strategy would be to return the entire sample. LotterySampling, with probability greater than $1-\delta$, returns all k heavy hitters with accurate frequencies (approximated by $1-\epsilon$ factor), plus some other elements whose frequency is at least $(1-\epsilon)f_{x_k}$, and it does so regardless of the stream, using more or less memory as the stream's distribution changes over time.

5 Conclusion

LotterySampling is an interesting and original algorithm mostly because it draws upon a simple but powerful enough concept: the lottery tickets.

For the Heavy Hitters problem, it is more or less on par with other existing alternatives (it satisfies the (δ, ϵ)-deficient framework) but it certainly does not outperform them, and it is less competitive on practical grounds. The Heavy Hitters version of LotterySampling is described and analyzed in this paper for completeness and to introduce the more important version for the Top-k problem, which is one of the main contributions of this paper.

For arbitrary and unknown streams, LotterySampling is able to solve the Top-k problem with probability greater than $1-\delta$, returning all the k most frequent elements with their frequencies approximated by a factor of $1-\epsilon$. It might also return a few more elements that are not among the top-k most frequent elements, but whose real frequency is at least $(1-\epsilon)f_{x_k}$. It does so with an upper bound of the expected memory usage independent of the stream distribution and its length N, since it will depend only on the unknown k^{th} largest (relative) frequency p_{x_k} and the parameters δ and ϵ used to determine the quality of the answer. Note that p_{x_k} determines the complexity of the stream, and that the sample of

LotterySampling will increase or decrease depending on whether p_{x_k} is decreases or increases. The time complexity for each occurrence in the stream is $\mathcal{O}(\log(|S|)$ in the worst case but most of the times it will be constant.

To the best of our knowledge, it is the first algorithm that has the ability to give strong probabilistic guarantees for the Top-k problem without previous knowledge of the stream. Note that for deterministic (exact) algorithms, in order to give the guarantees required by our proposed ϵ-deficient framework, the required space is at least linear in the input size [3], and it should be dependent on the stream distribution. However, we proved that LotterySampling removes this linear dependence on the input size by relaxing the constraints to a probabilistic framework.

Other state-of-the-art algorithms tackle the Top-k problem assuming previous knowledge of f_{x_k}, which changes radically the complexity of the problem, that becomes completely different respect to our problem definition. Despite LotterySampling, in the current form given here, is not competitive compared to well known alternatives in most practical situations (because some reasonable assumptions hold for almost all streams), we think that our initial result on LotterySampling and its core concepts will help to design new algorithms which are competitive in practical terms while retaining the probabilistic guarantees that other state-of-the-art alternatives lack of.

References

1. Bandi, N., Metwally, A., Agrawal, D., El Abbadi, A.: Fast data stream algorithms using associative memories. In: Proc. ACM Int. Conf. on Management of Data (SIGMOD 2007). pp. 247–256 (2007)
2. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. Theoretical Computer Science **312**(1), 3–15 (2004)
3. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. Proceedings of the VLDB Endowment **1**(2), 1530–1541 (2008)
4. Gibbons, P.B., Matias, Y.: New sampling-based summary statistics for improving approximate query answers. In: Proc. ACM Int. Conf. on Management of Data (SIGMOD 1998). pp. 331–342 (1998)
5. Lam, H., Calders, T.: Mining top-k frequent items in a data stream with flexible sliding windows. In: Proc. 16th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2010). pp. 283–292 (07 2010)
6. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proc. 28th Int. Conf. on Very Large Data Bases, (VLDB 2002). pp. 346–357. Morgan Kaufmann (2002)
7. Metwally, A., Agrawal, D., El Abbadi, A.: Efficient computation of frequent and top-k elements in data streams. In: Proc. 10th Int. Conf. on Database Theory (ICDT 2005). pp. 398–412 (2005)
8. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science **1**(2), 117–236 (2005)

A Experimental results

All the implementations of the algorithms and the test framework is publicly available in this repository under the name of `basic_lottery_sampling_hh` and `basic_lottery_sampling_top_k`.

A.1 Heavy Hitters

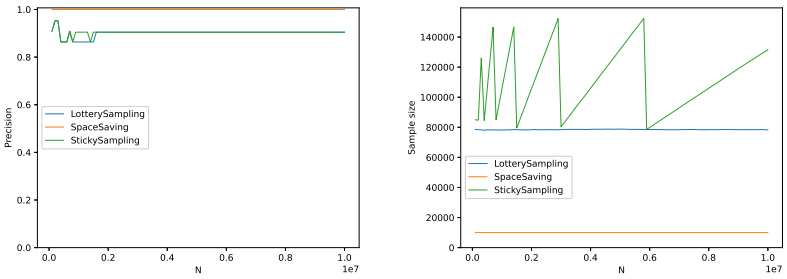

Fig. 1. Executing Heavy Hitter queries with $\phi = 0.001$, $\epsilon = 0.0001$ and $\delta = 0.1$ in a Zipfian stream of length $N = 10^7$ and $\alpha = 1.0001$.

A.2 Top-k

To showcase the properties of LotterySampling and SpaceSaving for the Top-k problem, we have used an artificial stream which we call ZipfNoiseZipf. It consists in a Zipfian distribution with $\alpha = 1.0001$, but during half of the stream (in the middle), some "noise" is added to modify the overall distribution of the stream while keeping the ranking. This "noisy" section consists in non-repeated elements. This way, the rank of all the elements is not modified, so the set of heavy hitters is always the same, but their relative frequencies decrease during the noisy section, or equivalently, the skewness of the stream decreases.

During that section, the sample size of LotterySampling increases to adapt to the change of distribution, since it has become a "harder" stream (note that p_{x_k} decreases). After the noisy section, the relative frequencies of the heavy hitters increase, and so does the skewness of the stream (as p_{x_k} recovers), making the sample size of LotterySampling to decrease. The result of this experiment appears in Figure 2.

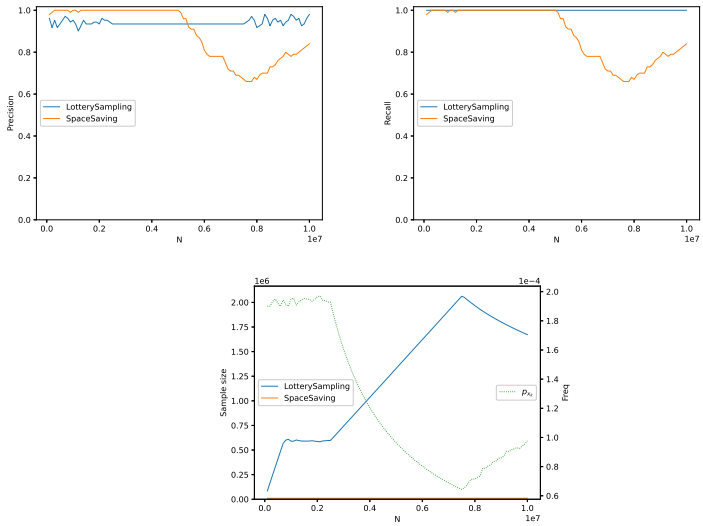

Fig. 2. ZipfNoiseZipf stream of length $N = 10^7$ and $\alpha = 1.0001$, executing Top-k queries with $k = 100$, $\epsilon = 0.1$ and $\delta = 0.1$. Sample size of SpaceSaving is $m = 10000$.