Streaming algorithms

1 Streaming Model

Streaming algorithms are memory-efficient algorithms that process a stream (i.e. a sequence) of data items in real time, to compute useful features of the data.

In order to motivate the streaming model of computation, consider the following scenario. Suppose we have a network router that is monitoring the internet traffic passing through it. The router encounters a stream of packets, and would like to estimate a few basic statistics associated with the internet traffic.

Let us flesh out the scenario with a few concrete numbers. Each network packet has a destination address – say an IPv6 address that is 128 bits long, i.e., there are 2^{128} distinct addresses possible. Over the course of a day, about 2^{40} packets make their way through the router. How can we estimate traffic statistics at the end of the day?

As a warmup, consider the easiest statistic to compute namely the number of packets seen during the day. In this case, we maintain a counter that is at least 40 bits long and increment the counter on seeing a packet.

Instead, suppose we want to estimate the total number of distinct IP addresses seen during the day. A naive algorithm would maintain an array "$isSeen[1, \ldots, 2^{128}]$" with one entry for each possible IP address, indicating whether the address is seen or not. All entries of $isSeen[]$ would be initialized to 0 at the beginning of the day. Every time a packet with address i is seen, the algorithm updates $isSeen[i]$ to 1. At the end of the day, one can determine the total number of distinct IP addresses seen.

It is clear that the memory needs of the above described naive algorithm make it impractical to implement. Any practical algorithm in this situation faces two severe restrictions:

- The space available at the router is too little to store all the different types of packets (say IP addresses) seen in the day.
- The router sees the stream of traffic (the input) only once, i.e., the algorithm can read the stream only once.

An algorithm designed with these restrictions is known as a *streaming algorithm*. Specifically, in the streaming model, the input is a stream of symbols $x_1, \ldots, x_n$ from some domain, say $\{1, \ldots, R\}$ for some R (think of R as polynomially large in n, say $R = n^3$). Typically, we will use Σ to denote the domain of each symbol, and size of the domain is denoted by $|\Sigma|$. The goal is to compute some function $f(x_1, \ldots, x_n)$ of the stream, such as the number of distinct elements in the stream.

The two critical restrictions on a streaming algorithm are that

- The space available to the algorithm is $poly(\log n)$
- The algorithm reads the input stream exactly once in the order $x_1, \ldots, x_n$.

Abstracting away all the data except the labels, we can think of our input as being a stream

$$s_1, s_2, \ldots s_n$$

where each $s_i \in \Sigma$ is a label, and Σ is the set of all possible labels (e.g. set of all possible 2^{128} IP addresses in our example). For a given stream $s_1, \ldots, s_n$, and for a label $a \in \Sigma$, we will call f_a the *frequency* of a in the stream, that is, the number of times a appears in the stream.

We will see streaming algorithms for the following three problems:

- Sample a random element of a stream. This is a rather useful primitive to estimate different quantities associated with a stream of data.
- Finding the number of distinct labels in the stream (e.g. finding how many different IP addresses the traffic flowing through the router emanated from)
- Finding *heavy hitters*, that is, labels for which f_a is large (e.g. find IP addresses that visited most often)

A streaming algorithm makes one pass through the data, and typically uses only $poly(\log n)$ bits of memory. For example, we will give a solution to the heavy hitter problem that uses $O((\log n)^2)$ bits of memory (in realistic implementations, the data structure requires only an array of size about 300-600, containing 32-bit integers) and solutions to the other problems that use $O(\log n)$ bits of memory (in realistic settings, the data structures are an array of size ranging from a few dozens to a few thousands 32-bit integers).

2 Probability Review

Since we are going to do a probabilistic analysis of randomized algorithms, let us review the handful of notions from discrete probability that we are going to use.

Suppose that A and B are two *events*, that is, things that may or may not happen. Then we have the *union bound*

$$\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$$

If A and B are *independent* then we also have

$$\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$$

Informally, a *random variable* is an outcome of a probabilistic experiment, which has various possible values with various probabilities. (Formally, it is a function from the sample space to the reals, though the formal definition does not help intuition very much.)

The *expectation* of a random variable X is

$$\mathbb{E}\, X = \sum_v \Pr[X = v] \cdot v$$

where v ranges over all possible values that the random variable can take.

We are going to make use of the following three useful facts about random variables:

- Linearity of expectation: if X and Y are two random variables, then $\mathbb{E}[X+Y] = \mathbb{E}\, X + \mathbb{E}\, y$, and if v is a number, then $\mathbb{E}[vX] = v \cdot \mathbb{E}\, X$.
- Product formula for independent random variables: If X and Y are independent, then $\mathbb{E}\, XY = (\mathbb{E}\, X) \cdot (\mathbb{E}\, Y)$
- Markov's inequality: If X is a random variable that is always ≥ 0, then, for every $t > 0$, we have
$$\Pr[X \geq t] \leq \frac{\mathbb{E}\, X}{t}$$

  Linearity of expectation and the product rule greatly simplify the computation of the expectation of random variables that come up in applications (which are often sums or products of simpler random variables). Markov's inequality is useful because once we compute the expectation of a random variable we are able to make high-probability statements about it. For example, if we know that X is a non-negative random variable of expectation 100, we can say that there is a $\geq 90\%$ probability that $X \leq 1,000$.

  Finally, the *variance* of a random variable X is

$$\mathbf{Var} X := \mathbb{E}(X - \mathbb{E}\, X)^2$$

and the *standard deviation* of a random variable X is $\sqrt{\mathbf{Var}X}$. The significance of this definition is that, if the variance is small, we can prove that X has, with high probability, values close to the expectation. That is, for every $t > 0$,

$$\Pr[|X - \mathbb{E}\,X| \geq t] = \Pr[(X - \mathbb{E}\,X)^2 \geq t^2] \leq \frac{\mathbf{Var}X}{t^2}$$

and, after the change of variable $t = c\sqrt{\mathbf{Var}X}$,

$$\Pr[|X - \mathbb{E}\,X| \geq c\sqrt{\mathbf{Var}X}] \leq \frac{1}{c^2}$$

so, for example, there is at least a 99% probability that X is within 10 standard deviations of its expectation. The above inequality is called Chebyshev's inequality. (There are a dozen alternative spellings for Chebyshev's name; you may have encountered this inequality before, spelled differently.)

If one knows more about X, then it is possible to say a lot more about the probability that X deviates from the expectation. In several interesting cases, for example, the probability of deviating by c standard deviations is exponentially, rather than polynomially, small in c^2. The advantage of Chebyshev's inequality is that one does not need to know anything about X other than its expectation and its variance.

Two final things about variance: if $X_1, \ldots, X_n$ are pairwise independent random variables, then

$$\mathbf{Var}[X_1 + \ldots + X_n] = \mathbf{Var}X_1 + \ldots + \mathbf{Var}X_n$$

and if X is a random variable whose only possible values are 0 and 1, then

$$\mathbb{E}\,X = \Pr[X = 1]$$

and

$$\mathbf{Var}X = \mathbb{E}\,X^2 - (\mathbb{E}\,X)^2 \leq \mathbb{E}\,X^2 = \mathbb{E}\,X = \Pr[X = 1]$$

3 Sampling

Perhaps, the simplest and most fundamental algorithm to estimate a statistic is *random sampling*. To illustrate the algorithm, let us consider a toy example.

Suppose we have a population of 300 voters participating in an election with two parties A and B. Our goal is to determine the fraction of the population voting for A. A simple sub-linear time algorithm would be to sample t voters out of 300

uniformly at random, and compute the fraction of the sampled voters who vote for A.

Let p denote the fraction of the population voting for A. Suppose we sample t people, each one independently and uniformly at random from the population. Let X_i denote the 0/1 random variable indicating whether the i^{th} sample, votes for A. The algorithm's estimate for p is given by,

$$\tilde{p} = \frac{1}{t} \sum_{i=1}^{t} X_i \ .$$

It is easy to check that,

$$\begin{aligned} \mathbb{E}[X_i] &= \Pr[X_i = 1] \cdot 1 + \Pr[X_i = 0] \cdot 0 \\ &= p \cdot 1 + (1 - p) \cdot 0 = p \ , \end{aligned}$$

and therefore $\mathbb{E}[\tilde{p}] = \frac{1}{t} \sum_i \mathbb{E}[X_i] = p$. So the expected value of the estimate $\tilde{p}$ is exactly equal to p (such an estimator is said to be *unbiased*).

The natural question to answer is, how many samples do we need in order to ensure that the estimate $\tilde{p}$ is within say 0.1 of the correct value p, with probability say 0.9. To answer this question, we will appeal to the following theorem.

Theorem 1 *(Chernoff/Hoeffding Bound) Suppose $X_1 \ldots, X_t$ are i.i.d random variables taking values in $\{0, 1\}$. Let $p = \mathbb{E}[X_i]$ and $\epsilon > 0$, then*

$$\Pr\left[\left|\frac{1}{t} \cdot \sum_{i=1}^{t} X_i - p\right| \geq \epsilon\right] \leq 2e^{-2\epsilon^2 t}$$

In particular, if we want to ensure that,

$$\Pr[\text{Estimate has an error } \geq \epsilon] \leq \delta$$

then we need to set $t = \lceil \frac{1}{2\epsilon^2} \log_e \left(\frac{2}{\delta}\right) \rceil$.

Notice that the number of samples needed to ensure that the estimate is within error ϵ with probability $1 - \delta$, is independent of the population size! In other words, to obtain a 0.1-approximate estimate with probability 0.9, we need to sample the same number of voters irrespective of whether the total population is 300 or 300 million. In this sense, the running time of random sampling algorithm is not just sub-linear in input, but independent of the input size!

4 Reservoir Sampling

Suppose we have a stream of items $s_1, \ldots, s_n$ and we would like to sample a uniformly random element of the stream. If we know the length of the stream apriori, then one can just pick a uniformly random index $j \in \{1, \ldots, n\}$ at the beginning of the algorithm. Then, the algorithm can output the j^{th} element of the stream when it arrives.

However, a streaming algorithm might not know of the total length of the stream (e.g. a router monitoring internet traffic doesn't know the number of packets in a day). Therefore, we will design a sampling algorithm that outputs a uniformly random element of the stream, at any point of its execution.

The idea is to maintain a *reservoir* that holds the current choice of the algorithm for the random sample. When the next element s of the stream arrives, the algorithm will place s in the reservoir (while discarding the sample that was already in it), with an appropriately chosen probability.

1: $reservoir \leftarrow s_1$
2: **for** $i = 2$ to n **do**
3: $\quad r \leftarrow$ random number between 1 to i
4: $\quad$ **if** $r = 1$ **then**
5: $\quad\quad reservoir \leftarrow s_i$
6: $\quad$ **end if**
7: **end for**
8: Output *reservoir*

Algorithm 1: Reservoir Sampling of One Element

Theorem 2 *The reservoir sampling algorithm outputs a uniformly random element of the stream*

Proof: To prove that the algorithm outputs a uniformly ranodm element of the stream, we will use induction on i. The inductive hypothesis is:

$$\textit{At the end of } i^{th} \textit{ iteration, for all } j < i, \Pr[reservoir = s[j]] = 1/i$$

Suppose the inductive hypothesis holds for $i-1$. Consider the ith iteration of algorithm. Then for any $j \leq i-1$,

$$\begin{aligned} &Pr[reservoir = stream[j] \text{ after iteration } i] \\ &\quad = \Pr[reservoir = stream[j] \text{ after iteration } i-1] \\ &\qquad\qquad \times \Pr[stream[j] \text{ is not replaced from } reservoir \text{ in iteration } i] \end{aligned}$$

By inductive hypothesis, the first term above is $\frac{1}{i-1}$. Since the element at the reservoir is replaced only when the random number r equals 1, the second term above is $1 - \frac{1}{i}$. Substituting, we get that

$$Pr[reservoir = stream[j] \text{ after iteration } i] = \frac{1}{i-1} \cdot \left(1 - \frac{1}{i}\right) \quad = \frac{1}{i-1} \cdot \left(\frac{i-1}{i}\right) = \frac{1}{i}$$

Finally for i^{th} element of the stream,

$$Pr[reservoir = stream[i] \text{ after iteration } i] = \Pr[r = 1] = \frac{1}{i}$$

□

To sample t elements of the stream independently (with replacement), one can just execute t parallel executions of the above algorithm over the stream. This will produce t uniformly random elements, possibly with repetition.

Suppose we want to sample t distinct elements of the stream (i.e., sampling without replacement), then one can maintain a reservoir of size t.

1: $reservoir[1, \ldots, t] \leftarrow s[1, \ldots, t]$
2: **for** $i = t + 1$ to n **do**
3: $\quad r \leftarrow$ random number between 1 to i
4: $\quad$ **if** $r \leq t$ **then**
5: $\quad\quad reservoir[r] \leftarrow s_i$
6: $\quad$ **end if**
7: **end for**
8: Output $reservoir[1, \ldots, t]$

Algorithm 2: Reservoir Sampling of t Elements without replacement

5 Counting Distinct Elements

Let us see a streaming algorithm to count the number of distinct elements. For sake of concreteness, let us consider the following setup of the problem: you are given a sequence of words $w_1, \ldots, w_n$ (say a really large piece of literature) and the goal is to estimate the number of distinct words in the sequence.

A trivial solution would involve scanning the words $w_1, \ldots, w_n$ once, while remembering at each point in time, all the words seen. In general, this algorithm requires $\Omega(n)$-bits of memory. Now, we will see an algorithm that uses sub-linear space, in fact, $poly(\log n)$ space.

Distinct Elements Algorithm We are now ready to describe a streaming algorithm for counting the number of distinct words. Let Σ denote the set of all possible words. First, we will describe an idealized algorithm to illustrate the main idea.

1: Pick a hash function $h : \Sigma \to [0, 1]$
2: Compute the minimum hash value $\alpha = \min_i h(w_i)$ by going over the stream
3: Output $\frac{1}{\alpha}$

Algorithm 3: Counting Distinct Elements

The algorithm finds the minimum hash value of a word in the stream, and outputs its inverse. It is easy to check that the algorithm can be implemented with $O(\log n)$ bits of space (after suitably discretizing the hash function).

To describe the main idea behind the algorithm, let us first make a strong assumption.

(Random Hash Assumption) *For each word $w \in \Sigma$, its hash value $h(w)$ is a uniformly random number in $[0, 1]$ independent of all other hash values*

Here is the intuition behind the algorithm. Suppose the stream $w_1, \ldots, w_n$ contains k different words, then the algorithm encounters k different hash values. If $r_1, \ldots, r_k$ are these k-different hash values, then the algorithm will output $\frac{1}{\min(r_1,\ldots,r_k)}$.

If $r_1, \ldots r_k$ are k independently chosen random numbers in $[0, 1]$ then we expect these numbers to be uniformly distributed in $[0, 1]$ and the smallest of them to be around $\frac{1}{k}$. Therefore, we can expect that the reciprocal of the minimum is approximately k – the number of distinct words.

More formally, one can show that

Lemma 3 *If there are k distinct elements in the stream then $\mathbb{E}[\min_i h(w_i)] = \frac{1}{k+1}$*

Feel free to skip the following proof, we include it here for sake of completeness. Proof: The above statement is equivalent to saying that if $r_1, \ldots, r_k$ are independently and uniformly distributed in $[0, 1]$ then,

$$\mathbb{E}[\min(r_1, \ldots, r_k)] = \frac{1}{k+1}$$

$$\begin{aligned}
\mathbb{E}[\min(r_1, \ldots, r_k)] &= \int_{r_1,\ldots,r_k} \min(r_1, \ldots, r_k) dr_1 dr_2 \ldots dr_k \\
&= \int_{r_1,\ldots,r_k} \left(\int_{r_{k+1}=0}^{1} 1[r_{k+1} \leq \min(r_1, \ldots, r_k)] dr_{k+1} \right) dr_1 dr_2 \ldots dr_k \\
&= \Pr_{r_1,\ldots,r_k,r_{k+1}} [r_{k+1} \leq \min(r_1, \ldots, r_k)]
\end{aligned}$$

where $r_1, \ldots, r_k, r_{k+1}$ are uniformly random elements in $[0,1]$. But, $r_{k+1} \leq \min(r_1, \ldots, r_k)$ if and only if $r_{k+1} = \min(r_1, \ldots, r_{k+1})$. The claim follows by observing that $r_{k+1} = \min(r_1, \ldots, r_{k+1})$ with probability exactly $\frac{1}{k+1}$, since any of the $k+1$ elements $\{r_1, \ldots, r_{k+1}\}$ is equally likely to be the smallest element. $\square$

Here is a simple calculation showing that there is at least 60% probability that the algorithm achieves a constant-factor approximation. Suppose that the stream has k distinct labels, and call $r_1, \ldots, r_k$ the k random numbers in $[0,1]$ corresponding to the evaluation of h at the distinct labels of the stream. Then

$$\begin{aligned}
\Pr\left[\text{Algorithm}'\text{s output } \leq \frac{k}{2}\right] &= \Pr\left[\min\{r_1, \ldots, r_k\} \geq \frac{2}{k}\right] \\
&= \Pr\left[r_1 \geq \frac{2}{k} \wedge \ldots \wedge r_k \geq \frac{2}{k}\right] \\
&= \Pr\left[r_1 \geq \frac{2}{k}\right] \cdot \ldots \cdot \Pr\left[r_k \geq \frac{2}{k}\right] \\
&= \left(1 - \frac{2}{k}\right)^k \\
&\leq e^{-2} \leq .14
\end{aligned}$$

and

$$\begin{aligned}
\Pr\left[\text{Algorithm}'\text{s output } \geq 4k\right] &= \Pr\left[\min\{r_1, \ldots, r_k\} \leq \frac{1}{4k}\right] \\
&= \Pr\left[r_1 \leq \frac{1}{4k} \vee \ldots \vee r_k \leq \frac{1}{4k}\right] \\
&\leq \sum_{i=1}^{k} \Pr\left[r_i \leq \frac{1}{4k}\right] \\
&= \frac{1}{4}
\end{aligned}$$

so that

$$\Pr\left[\frac{k}{2} \leq \text{Algorithm}'\text{s output } \leq 4k\right] \geq .61$$

There are various ways to improve the quality of the approximation and to increase the probability of success.

One of the simplest ways is to keep track not of the smallest hash value encountered so far, but the t *distinct labels with the smallest hash values*. This is a set of labels

and values that can be kept in a data structure of size $O(t \log |\Sigma|)$, and processing an element from the stream takes time $O(\log t)$ if the labels are put in a priority queue. Then, if tsh is the t-th smallest hash value in the stream, our estimate for the number of distinct values is $\frac{t}{tsh}$.

The intuition is that, as before, if we have k distinct labels, their hashed values will be k random points in the interval $[0, 1]$, which we would expect to be uniformly spaced, so that the t-th smallest would have a value of about $\frac{t}{k}$. Thus its inverse, multiplied by t, should be an estimate of k.

Why would it help to work with the t-th smallest hash instead of the smallest? The intuition is that it takes only one outlier to skew the minimum, but one needs to have t outliers to skew the t-th smallest, and the latter is a more unlikely event.

6 Pseudorandom functions

The above analysis elucidates the central idea underlying our streaming algorithm. However, it crucially relies on the assumption that the algorithm has access to a hash function h that is a *uniformly random* function from $\Sigma \to [0, 1]$. Therefore, there are a few roadblocks to implementing the algorithm as described.

First, in an actual implementation, the output of a hash function h can only have a finite range, say $\{1, \ldots, R\}$ for some positive integer R. By setting $h'(x) = h(x)/R$, we can obtain a hash function that takes a discrete set of values in $[0, 1]$. The analysis of the streaming algorithm can be modified to take this discretization into account. Specifically, one can show that discretization introduces an additive error of at most ϵ if we use a sufficiently large range R (say $R > n/\epsilon$). Hence, a *uniformly random hash function* $h : \Sigma \to \{1, \ldots, R\}$ would suffice to implement the algorithm.

To be sure, let us understand how one could in principle generate a *uniformly random* hash function $h : \Sigma \to \{1, \ldots, R\}$. Let us assume that the algorithm has access to uniformly random bits, as many as it needs. For each $x \in \Sigma$, we can pick a random number $\gamma \in \{1, \ldots, R\}$ and set $h(x) = \gamma$, and construct a look-up table for the function h. This would constitute a *uniformly random function* h.

However, a streaming algorithm would not be able to store such a lookup table that has $\Omega(|\Sigma|)$ entries. Furthermore, by virtue of being *uniformly random*, the function h cannot be represented in any way other than a lookup table. Consequently, a streaming algorithm cannot use a *uniformly random* hash function. Instead, we will have our streaming algorithm pick a hash function at random from a small family of functions.

Formally, a *hash family* $\mathcal{H}$ is just a family of functions each with domain Σ and range $\{1, \ldots, R\}$. To pick a random hash function from family $\mathcal{H} = \{h_1, \ldots, h_{|\mathcal{H}|}\}$, an algorithm needs to sample a random number $r \in \{1, \ldots, |\mathcal{H}|\}$ and then use the

function $h := h_r$. To store the hash function h, it is enough to store the index r which takes $O(\log |\mathcal{H}|)$ bits.

For example, suppose $\mathcal{H}$ is the set of all possible functions from the domain to the range. Then, selecting a hash function h from $\mathcal{H}$ corresponds to sampling a *uniformly random* function. In this case, the function h is highly random, but the size of $\mathcal{H}$ and consequently the memory needed to store h is large. At the other extreme, suppose $\mathcal{H}$ consists of a fixed pair of functions $\{h_0, h_1\}$. A function h chosen from hash family $\mathcal{H}$ needs just one bit to memorize, but is very far from random. In fact, for each x, $h(x)$ takes only one of at most two values.

Intuitively, we need a small hash family $\mathcal{H}$ such that a function h chosen randomly from $\mathcal{H}$ is *sufficiently random looking* for the streaming algorithm. In particular, we would like to be able to prove the correctness of the streaming algorithm when it uses a hash function from the family $\mathcal{H}$. It turns out that all we need from the hash family is the property of *pairwise independence*.

Definition 4 *(Pairwise independent hash family)*

A family of functions $\mathcal{H} = \{h_1, \ldots, h_M\}$ from a domain Σ to a range R, is said to be a pairwise-independent hash family if the following holds: If we pick a hash function h at random from $\mathcal{H}$, then on any pair of inputs $x, y \in \Sigma$, the behaviour of h exactly mimics that of a completely random function. Formally, for all $x \neq y \in \Sigma$ and $i, j \in R$,

$$\Pr_{h \in \mathcal{H}}[h(x) = i \wedge h(y) = j] = \frac{1}{|R|^2}$$

The above definition also implies that each hash value is uniformly distributed by itself, i.e., For every $a \in \Sigma$, and for every $r \in \{1, \ldots, R\}$

$$\Pr[h(a) = r] = \frac{1}{R}$$

However a hash function chosen from a pairwise-independent hash family only looks random on two inputs at a time. If we consider three different hashes $h(x), h(y)$ and $h(z)$ simultaneously, then the three values might not appear random at all.

Here is a simple and elegant example of a pairwise-independent hash family that is of small size. Fix a prime number p. For each $a \in Z_p$ and $b \in \mathbb{Z}_p$, define $h_{a,b} : \mathbb{Z}_p \to \mathbb{Z}_p$ as,

$$h_{a,b}(x) = a \cdot x + b \mod p \ ,$$

and the hash family $\mathcal{H} = \{h_{a,b} | a \in Z_p, b \in \mathbb{Z}_p\}$. $\mathcal{H}$ is a universal hash family such that any function in the family can be represented in $O(\log p)$ bits by storing two integers $a, b \in Z_p$.

It is a good excercise to show that the above defined family is indeed pairwise independent. There are many other constructions of pairwise independent hash families (see Excercise 1.29 in textbook).

In the next section, we include a proof of correctness of the streaming algorithm for distinct elements while using a pairwise independent hash family.

6.1 * Rigorous Analysis of the "t-th Smallest" Algorithm

Let us sketch a more rigorous analysis. These calculations are a bit more complicated than the rest of the content of this lecture, so it is OK to skip them, we include it just for sake of completeness

We will see that we can get an estimate that is, with high probability, between $k - \epsilon k$ and $k + \epsilon k$, by choosing t to be of the order of $1/\epsilon^2$. For example, choosing $t = 30/\epsilon^2$ there is at least a 93% probability of getting an ϵ-approximation.

For concreteness, we will see that, with $t = 3,000$, we get, with probability at least 93%, an error that is at most 10%. As before, we let $r_1, \ldots, r_k$ be the hashes of the k distinct labels in the stream. We let tsh be the t-th smallest of $r_1, \ldots, r_k$.

$$\begin{aligned}\Pr[\text{Algorithm}'\text{s output } \geq 1.1k] &= \Pr\left[tsh \leq \frac{t}{1.1k}\right] \\ &= \Pr\left[\left(\#i : r_i \leq \frac{t}{1.1k}\right) \geq t\right]\end{aligned}$$

Now let's study the number of is such that $r_i \leq t/(1.1k)$, and let's give it a name

$$N := \#i : r_i \leq \frac{t}{1.1k}$$

This is a random variable whose expectation is easy to compute. If we define S_i to be 1 if $r_i \leq t/(1.1k)$ and 0 otherwise, then $N = \sum_i S_i$ and so

$$\mathbb{E}\, N = \sum_i \mathbb{E}\, S_i = \sum_i \Pr\left[r_i \leq \frac{t}{1.1k}\right] = k \cdot \frac{t}{1.1k} = \frac{t}{1.1}$$

The variance of N is also easy to compute.

$$\mathbf{Var} N = \sum_i \mathbf{Var} S_i \leq k \cdot \frac{t}{1.1k} \leq t$$

So N has an average of about $.91t$ and a standard deviation of less than $\sqrt{t}$, so by Chebyshev's inequality

$$\begin{aligned}\Pr[\text{Algorithm}'\text{s output } \geq 1.1k] &= \Pr[N \geq t] \\ &= \Pr[(N - \mathbb{E}\, N) \geq t - t/1.1] \\ &\leq \frac{\mathbf{Var} N}{(t/11)^2} \\ &= \frac{121}{t} \\ &= \frac{121}{3000} \leq 4.1\%\end{aligned}$$

Similarly,

$$\begin{aligned}\Pr[\text{Algorithm}'\text{s output } \leq .9k] &= \Pr\left[tsh \geq \frac{t}{.9k}\right] \\ &= \Pr\left[\left(\#i : r_i \leq \frac{t}{.9k}\right) \leq t\right]\end{aligned}$$

and if we call

$$N := \#i : r_i \leq \frac{t}{.9k}$$

We see that

$$\begin{aligned}\mathbb{E}\, N &= \frac{t}{.9} \\ \mathbf{Var} N &\leq t\end{aligned}$$

and

$$\begin{aligned}\Pr[\text{Algorithm}'\text{s output } \leq .9k] &= \Pr[N \leq t] \\ &= \Pr\left[\mathbb{E}\, N - N \geq \frac{t}{.9} - t\right] \\ &\leq \frac{\mathbf{Var} N}{(t/9)^2} \\ &= \frac{81}{t} \\ &= \frac{81}{3000} = 2.7\%\end{aligned}$$

So all together we have

$$\Pr[.9k \leq \text{Algorithm}'\text{s output } \leq 1.1k] \geq 93\%$$

Notice that the above analysis of the t-th smallest algorithm, does not need h to be random function $h : \Sigma \to \{1/N, \ldots, 1\}$: we just need the calculation of the expectation and variance of the number of labels in a certain set whose hash is in a certain range. For this, we just need, for every $a \neq b$, the values $h(a)$ and $h(b)$ to be independently distributed. Indeed, even if there was a small correlation between the distribution of $h(a)$ and $h(b)$, this could also be absorbed into the error calculations.

7 The Heavy Hitters Problem

A common special case of the heavy-hitters problem that you might be familiar is that of finding an element that appears a majority of times, i.e., an element a such that its frequencey $f_a > \frac{n}{2}$ in a stream of length n. The solution is an algorithm that is deterministic and uses only two variables (and $\log \Sigma + \log n$ bits of memory), but its analysis relies on the fact that we are interested in finding a label occurring a majority of the times. Suppose, instead, that we are interested in finding all labels, if any, that occur at least $.3n$ times. In this lecture, we will show that this requires $\Omega(\min\{|\Sigma|, n\})$ bits of memory. The data structure that we present in this section, however, is able to do the following using $O((\log n)^2)$ memory: come up with a list that includes *all* labels that occur at least $.3n$ times and which includes, with high probability, none of the labels that occur less than $.2n$ times. Of course .2 and .3 could be replaced by any other two constants.

Even more impressively, every time the algorithm sees an item in the stream, it is able to approximate the number of times it has seen that element so far, up to an additive error of $.1n$, and, again, .1 could be replaced by any other positive constant.

The algorithm is called Count-Min, or Count-Min-Sketch, and it has been implemented in a number of libraries.

The algorithm relies on two parameters ℓ and B. We choose $\ell = 2 \log n$ and $B = 20$. In general, if we want to be able to approximate frequencies up to an additive error of ϵn, we will choose $B = 2/\epsilon$.

The following version of the algorithm reports an approximation of the number of times it has seen the label so far for each label of the stream that it processes:

- Initialize an $\ell \times B$ array M to all zeroes
- Pick ℓ random functions $h_1, \ldots, h_\ell$, where $h_i : \Sigma \to \{1, \ldots, B\}$

- while not end-of-stream:
  - read a label x from the stream
  - for $i = 1$ to ℓ:
    - $M[i, h_i(x)] + +$
  - print "estimated number of times", x, "occurred so far is", $\min_{i=1,\ldots,\ell} M[i, h_i(x)]$

And the following version of the algorithm constructs a list that, includes all the labels that occur $\geq .3n$ times in the stream and, with high probability, includes none of the labels that occur $< .3n - \frac{2n}{B}$ times in the stream. (If $B = 20$, then $.3n - \frac{2n}{B} = .2n$.)

- Initialize an $\ell \times B$ array M to all zeroes
- Initialize L to an empty list
- Pick ℓ random functions $h_1, \ldots, h_\ell$, where $h_i : \Sigma \to \{1, \ldots, B\}$
- while not end-of-stream:
  - read a label x from the stream
  - for $i = 1$ to ℓ:
    - $M[i, h_i(x)] + +$
  - if $\min_{i=1,\ldots,\ell} M[i, h_i(x)] > .3n$ add x to L, if not already present
- return L

Let us analyze the above algorithm. The first observation is that, when we see a label a, we increase all the ℓ values

$$M[1, h_1(a)], M[2, h_2(a)], \ldots, M[\ell, h_\ell(a)]$$

and so, at the end of the stream, having done the above f_a times, it follows that all the above values are at least f_a, and so

$$f_a \leq \min_{i=1,\ldots,\ell} M[i, h_i(a)]$$

In particular, this means that if a is a label such that $f_a \geq .3n$ then, by the last time we see an occurrence of a, it must be that $\min_{i=1,\ldots,\ell} M[i, h_i(a)] > .3n$, and so, in the second algorithm, we see that a is certainly added to the list.

The problem is that $\min_{i=1,\ldots,\ell} M[i, h_i(a)]$ could be much bigger than f_a, and so the first algorithm could deliver a poor approximation and the second algorithm could

add some "light hitters" to the list. We need to prove that this failure mode has a low probability of happening.

First of all, we see that, for a fixed choice of the functions h_i,

$$M[i, h_i(a)] = f_a + \sum_{b \neq a : h_i(b) = h_i(a)} f_b$$

Now let's compute the expectation of $M[i, h_i(a)]$ over the random choice of the function h_i. We see that it is

$$\begin{aligned}
\mathbb{E}\, M[i, h_i(a)] &= f_a + \sum_{b \neq a} \Pr[h_i(a) = h_i(b)] \cdot f_b \\
&= f_a + \frac{1}{B} \sum_{b \neq a} f_b \\
&\leq f_a + \frac{n}{B}
\end{aligned}$$

where we use the fact that, for a random function $h_i : \Sigma \to \{1, \ldots, B\}$, the probability that $h_i(a) = h_i(b)$ is exactly $\frac{1}{B}$, and the fact that $\sum_{b \neq a} f_b = n - f_a \leq n$.

So far, we have proved that the content of $M[i, h_i(a)]$ is always at least f_a and, in expectation, is at most $f_a + \frac{n}{B}$. Let us know translate this statement about expectations to a statement about probabilities.

Applying Markov's inequality to the (non-negative!) random variable $M[i, h_i(a)] - f_a$, we have

$$\Pr\left[M[i, h_i(a)] > f_a + \frac{2n}{B}\right] \leq \frac{1}{2}$$

and, using the independence of the h_i,

$$\begin{aligned}
&\Pr\left[\min_{i=1\ldots,\ell} M[i, h_i(a)] > f_a + \frac{2n}{B}\right] \\
=&\Pr\left[\left(M[1, h_1(a) > f_a + \frac{2n}{B}\right) \wedge \ldots \wedge \left(M[\ell, h_\ell(a) > f_a + \frac{2n}{B}\right)\right] \\
=&\Pr\left[M[1, h_1(a) > f_a + \frac{2n}{B}\right] \cdot \ldots \cdot \Pr\left[M[\ell, h_\ell(a) > f_a + \frac{2n}{B}\right] \\
\leq&\frac{1}{2^\ell}
\end{aligned}$$

So, if we choose $B = 20$ and $\ell = 2 \log n$, we have that the estimate $\min_i M[i, h_i(a)]$ is between f_a and $f_a + .1n$, except with probability at most $\frac{1}{n^2}$. In particular, there is a probability at least $1 - 1/n$, that we get a good estimate n times in a row.

In the analysis of the heavy hitter algorithm, we just use the fact that for every two distinct labels a, b, we have

$$\Pr[h(a) = h(b)] = \frac{1}{B} ,$$

a property that is satisfied by a pairwise-independent hash family.

Summary

We have the following algorithmic guarantees for the problems that we have studied:

- Heavy hitters: for every fixed threshold $0 < t < 1$, and approximation parameter ϵ, given a stream of n elements of Σ, we can construct, using space $O(\epsilon^{-1} \cdot (\log n) \cdot (\log n + \log |\Sigma|))$, a list that:
  - With probability 1, contains all the labels that occur $\geq t \cdot n$ times in the stream
  - With probability $\geq 1 - 1/n$, contains no label that occurs $\leq (t - \epsilon) \cdot n$ times in the stream
- Distinct elements: for every approximation parameter ϵ, given a stream of n elements of Σ, we can compute a number that is, with probability $> 90\%$, between $k - \epsilon k$ and $k + \epsilon k$, where k is the number of distinct elements in the stream, using space $O(\epsilon^{-2} \cdot (\log n + \log |\Sigma|))$

8 Memory lower bounds

We will now prove memory lower bounds for streaming algorithms that are deterministic and exact. To prove memory lower bounds, we will show that if there was an exact deterministic algorithm that uses $o(\min\{|\Sigma|, n\})$ memory and solves the heavy hitters problem or counts the number of distinct elements, then there would be an algorithm that is able to compress any L-bit file to a $o(L)$-bit compression. The latter is clearly impossible, and so sub-linear memory deterministic exact streaming algorithms cannot exist.

In the following, a "compression algorithm" is any injective function C that maps bit strings to bit strings. The following is well known.

Theorem 5 *There is no injective function C that maps all L-bit strings to bit strings of length $\leq L-1$.*

PROOF: There are 2^L bit strings of length L and only $2^L - 1$ bit strings of length $\leq L-1$. $\square$

The lower bound for counting distinct elements follows from the lemma below.

Lemma 6 *Suppose that there is a deterministic exact algorithm for counting distinct elements that uses $o(\min\{|\Sigma|, n\})$ bits of memory to process a stream of n elements of Σ.*

Then there is a compression algorithm that maps L-bit strings to bits strings of length $o(L)$.

Since the conclusion is false, the premise is also false, and so every deterministic exact algorithm for counting distinct elements must use memory $\Omega(\min\{|\Sigma|, n\})$.

Here is how we prove the lemma: given a string $b_1, \ldots, b_L$ of L bits that we want to compress, we define Σ as the set

$$\{(1,0),(1,1),(2,0),(2,1),\ldots,(L,0),(L,1)\}$$

and we consider the stream

$$(1,b_1),(2,b_2),\ldots,(L,b_L)$$

We run our hypothetical streaming algorithm on the above stream, and we take *the state of the algorithm at the end of the computation* as our compression of the string $b_1, \ldots, L$. Note that $n = L$ and $\Sigma = 2L$, so the state of the algorithm is $o(L)$ bits.

Why is this a valid compression? Using the state of the algorithm, we can find what is the number of distinct elements in the stream

$$(1,b_1),(2,b_2),\ldots,(L,b_L),(1,0)$$

just by restarting the algorithm and presenting $(1,0)$ as an additional input. Now, the number of distinct elements will be L if $b_1 = 0$ and $L+1$ if $b_1 = 1$. So we have found the first bit of the string. Similarly, for each i, we can find out the number of distinct elememnts in

$$(1,b_1),(2,b_2),\ldots,(L,b_L),(i,0)$$

and so we can reconstruct the whole string.

Similarly, one can rule out deterministic algorithms to compute heavy hitters. We include the proof for completeness, feel free to skip it.

Lemma 7 *Suppose that there is a deterministic algorithm f that uses $o(\min\{|\Sigma, n\})$ bits of memory to process a stream of n elements of Σ and outputs a list that contains all, and only, the labels that occur at least $.3n$ times in the stream.*

Then there is a compression algorithm that maps L-bit strings to bits strings of length $o(L)$.

This time, if we want to compress a string $b_1, \ldots, b_L$, we use

$$\Sigma = \{(1,0), (1,1), (2,0), 2, 1), \ldots, (L,0), (L,1), \perp\}$$

and our compression is the state of the algorithm after processing

$$(1, b_1), (2, b_2), \ldots, (L, b_L), \perp, \ldots, \perp$$

where $\perp$ is repeated $.4L + 1$ times.

The key observation is that, for every i, the stream

$$(1, b_1), (2, b_2), \ldots, (L, b_L), \perp, \ldots, \perp, (i, 0), \ldots, (i, 0)$$

where (i,0) is repeated $.6L - 1$ times, is of length $n = 2L$, and $(i, 0)$ appears $.6L = .3n$ times if $b_i = 0$ and only $.6L - 1 < .3n$ if $b_i = 1$. Thus $(i, 0)$ will be in the output of the heavy hitter algorithm in the first case, and not in the second. Repeating this for every i allows us to reconstruct the string.