Streaming algorithms

In this lecture and the next one we study memory-efficient algorithms that process a stream (i.e. a sequence) of data items and are able, in real time, to compute useful features of the data.

Consider a network router that is monitoring the internet traffic that passes through it. The router encounters a stream of packets, and would like to estimate some statistics associated with the internet traffic.

Here the algorithm has to face two severe restrictions:

- The space available at the router is too little to store all the 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 $\{1, \ldots, N\}$. The goal is to compute some function $f(x_1, \ldots, x_n)$. However, the space available to the algorithm is $poly(\log n)$ and 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 $s_i \in [N]$ is a label, and $[N]$ is the set of all possible labels (e.g. all product identifiers, or all IP addresses). For a given stream $s_1, \ldots, s_n$, and for a label $a \in [N]$, 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 be interested in 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)

All three problems have simple solutions in which we store the entire stream. We can store a list of all the distinct labels we have seen, and for each of them also store the number of occurrences. That is, the data structure would contain a pair (a, f_a) for every label a that appears at least once in the stream. Such a data structure can be maintained using $O(k \cdot (\log n + \log |\Sigma|)) \leq O(n \cdot (\log n + \log |\Sigma|))$ bits of memory. Alternatively, one could have an array of size $|\Sigma|$ storing f_a for every label a, using $O(|\Sigma| \cdot \log n)$ bits of memory. If the labels are IPv6 addresses, then $\Sigma = 2^{128}$, so the second solution is definitely not feasible; in a setup in which a stream might contain hundreds of millions of items or more, even a data structure of the first type is rather large.

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).

1 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*

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

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

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

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

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

2   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 ith 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,

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

4

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!

3 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 jth 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 ← s₁
2: for i = 2 to n do
3:     r ← random number between 1 to i
4:     if r = 1 then
5:         reservoir ← sᵢ
6:     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:

At the end of ith 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 $j \leq i - 1$,

$Pr[reservoir = stream[j]$ after iteration $i]$

$\qquad = \Pr[reservoir = stream[j]$ after iteration $i - 1]$

$\qquad\qquad\qquad \times \Pr[stream[j]$ is not replaced from $reservoir$ in iteration $i]$

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

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

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:    $r \leftarrow$ random number between 1 to i
4:    **if** $r \leq t$ **then**
5:       $reservoir[r] \leftarrow s_i$
6:    **end if**
7: **end for**
8: Output $reservoir[1, \ldots, t]$

Algorithm 2: Reservoir Sampling of t Elements without replacement

4 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 $\frac{1}{\min(r_1, \ldots, r_k)}$.

7

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* $\mathbb{E}\left[\min_i h(w_i)\right] = \frac{1}{k+1}$

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,

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

$$\mathbb{E}[\min(r_1, \ldots, r_k)] = \int_{r_1, \ldots, r_k} \min(r_1, \ldots, r_k) dr_1 dr_2 \ldots dr_k$$

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

$$= \Pr_{r_1, \ldots, r_k, r_{k+1}} [r_{k+1} \leq \min(r_1, \ldots, r_k)]$$

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

$$\Pr\left[\text{Algorithm's output} \leq \frac{k}{2}\right] = \Pr\left[\min\{r_1, \ldots, r_k\} \geq \frac{2}{k}\right]$$

$$= \Pr\left[r_1 \geq \frac{2}{k} \wedge \ldots \wedge r_k \geq \frac{2}{k}\right]$$

$$= \Pr\left[r_1 \geq \frac{2}{k}\right] \cdot \ldots \cdot \Pr\left[r_k \geq \frac{2}{k}\right]$$

$$= \left(1 - \frac{2}{k}\right)^k$$

$$\leq e^{-2} \leq .14$$

and

$$\Pr\left[\text{Algorithm's output } \geq 4k\right] = \Pr\left[\min\{r_1, \ldots, r_k\} \leq \frac{1}{4k}\right]$$

$$= \Pr\left[r_1 \leq \frac{1}{4k} \vee \ldots \vee r_k \leq \frac{1}{4k}\right]$$

$$\leq \sum_{i=1}^{k} \Pr\left[r_i \leq \frac{1}{4k}\right]$$

$$= \frac{1}{4}$$

so that

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

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.

5   Pseudorandom functions

While the above analysis captures the basic intuition behind the algorithm, there is a major flaw in the argument. Specifically, the analysis crucially relied on the assumption that the hash values of different words are independent uniformly distributed random values in $[0, 1]$.

This assumption is too strong to hold for any reasonable hash function. Random objects are incompressible, so we would need $\Omega(|\Sigma|)$ bits to store such functions.

Even worse, a random function $h : \Sigma \to [0, 1]$ cannot be stored at all, because a random real number cannot be represented using a finite number of bits.

First, the analysis of the algorithm for the number of distinct elements, we can assume that the function maps uniformly to a discretized set

$$\left\{ \frac{1}{N}, \frac{2}{N}, \cdots, 1 \right\}$$

instead of $[0, 1]$, and if $N > n/\epsilon$ this introduces at most an additional ϵ error in the calculations.

It turns out that one can perform a different analysis of the algorithm (included in the next section) that does not need all the hash values to be completely independent and uniformly random. Instead, we just need the following property to hold.

Definition 4 *(Universal Hash Family a.k.a. 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 universal 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 $i, j \in R$,

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

Notice that the above definition also each hash value output to be uniformly distributed in itself, i.e., For every $a \in \Sigma$, and for every $r \in R$

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

However a hash function chosen from a universal 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.

A space-efficient streaming algorithm needs hash functions that have a small representation, say using only $poly(\log n)$ bits of memory. Here is a simple and elegant example of a universal hash family that uses small space. 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 $a, b \in Z_p$.

Using a slightly more involved analysis, one can show that the distinct elements algorithm still works with a pairwise independent hash family. We include the analysis below for sake of completeness.

5.1 * 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 $r_1, \ldots, r_k$.

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

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

11

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

Similarly,

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

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

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

12

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.