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.

Streaming algorithms are useful in any problem domain, including graph algorithms, but we will restrict to the following important family of problems: we get a sequence of data items (for example sales records, or web hits), and each data item has a data field of interest, which we will call a "label" (for example, the product id of the item that was sold, the IP address that we get a hit from) and we want to compute various statistics about the sequence of labels.

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

$$x_1, x_2, \dots x_n$$

where each $x_i \in \Sigma$ is a label, and Σ is the set of all possible labels (e.g. all product identifiers, or all IP addresses). For a given stream $x_1, \dots, x_n$, 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 be interested in the following three problems:

- Finding *heavy hitters*, that is, labels for which f_a is large (e.g. find best-selling products, or IP addresses that visit our site most often)
- Finding the number of distinct labels in the stream (e.g. finding how many different products we have sold, or how many unique visitors we have)
- Finding the sum-of-squares quantity $\sum_{a\in\Sigma} f_a^2$, which is usually denoted as F_2, and also called the *second frequency moment* of the stream

It should be self-evident that the first two problems are important.

To understand the quantity F_2, consider two extreme cases that we can have in a stream of n items. If all the n labels in the stream are different, then we have $\sum_{a\in\Sigma} f_a^2 = n$, because we have n frequencies that are 1 and all the others are zero. If all the n labels are identical, then $\sum_{a\in\Sigma} f_a^2 = n^2$. In all other cases, the value of F_2 for a stream of n items will always be between n and n^2, and smaller values correspond to streams with several distinct labels each occurring not too often, and large values correspond to streams with few distinct labels occurring often. So F_2 is a one-parameter measure of how "diversified" is the stream.

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.

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).

Those solutions will be *randomized* and will only provide *approximate solutions*. Both features are necessary: it is possible to prove that randomized exact algorithms and deterministic approximate algorithms must use an amount of memory of the same order as the trivial solution of storing all the values $f(a)$. We will not prove these impossibility results, but next week we will prove that any deterministic *and* exact algorithm for each of the above three problems must use $\Omega(\min\{|\Sigma|, n\})$ bits of memory.

Today we see algorithms for the heavy hitters and distinct elements problems. Next week we will see the sum-of-squares algorithm.

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*

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

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

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

2 The Heavy Hitters Problem

From Problem 5 in HW1, you may remember the problem of finding a label such that $f_a > \frac{n}{2}$ 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. Next week, 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 $\ell$ 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 $\ell$:
        * $M[i, h_i(x)] + +$
    – 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 $\ell$ 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 $\ell$:
        * $M[i, h_i(x)] + +$
    – 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.

3 Counting Distinct Elements

Here the algorithm is a lot simpler. We pick a random function $h : \Sigma \to [0,1]$ (we will see later how to appropriately discretize this choice so that the function can be stored in memory), and, for every item x in the stream, we evaluate $h(x)$, keeping track of the smallest hash value obtained so far. If min is the minimum hash value seen at the end of the stream, then $\frac{1}{\min}$ is our estimate of the number of distinct elements in the stream.

The intuition for the algorithm is the following: suppose that the stream has k distinct labels. Then when we evaluate h at every element of the stream, we are evaluating h at k distinct inputs, although some inputs are maybe repeated several times. If h is a random function, we are getting k random values in the interval $[0,1]$. Intuitively, we would expect k random numbers selected in the interval $[0,1]$ to be evenly distributed, and so the minimum of these numbers should be about $1/k$. Thus, the *inverse of the minimum* should be around k.

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

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

and

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

so that

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

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.

3.1 * 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 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 $r_1, \ldots, r_k$.

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

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

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

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

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

So all together we have

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