Streaming algorithms – continued

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 $O(\epsilon^{-2} \cdot (\log n + \log |\Sigma|))$

1 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 1 *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 2 *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 3 *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.