Some notes on streaming algorithms – continued

Today we complete our quick review of streaming algorithms. We show how to approximate F_2, how to generate pseudo-random functions that have a compact representation. This will show that, for every fixed approximation parameter $\epsilon > 0$, we can solve the heavy hitters in $O((\log n) \cdot (\log n + \log |\Sigma|))$ space, number of distinct elements in $O(\log n + \log |\Sigma|)$ space and F_2 is $O(\log n + \log |\Sigma|)$ space.

We will then prove memory lower bounds for streaming algorithms that are deterministic and exact.

1 Approximating F_2

If we have a stream $x_1, \ldots, x_n$ and f_a is the number of times a appears in the stream, then we define the F_2 parameter of the stream as

$$F_2 := \sum_{a \in \Sigma} f_a^2$$

As we discussed last week, this parameter is always between n and n^2 and it measures how "diverse" is the stream: roughly speaking, low values correspond to a stream in which there are many elements occurring a small number of times and high values correspond to streams in which few elements occur a large number of times.

The basic idea of the algorithm is to pick a random function $h : \Sigma \to \{-1, +1\}$ and, given a stream $x_1, \ldots, x_n$, compute

$$Z := \sum_i h(x_i)$$

which we can do easily by initializing $Z := 0$ and then adding to Z the hash of the current stream element at every step.

Our estimate for the sum-of-square parameter is Z^2.

Note that

$$Z = \sum_{a \in \Sigma} f_a h(a)$$

and so

$$\begin{aligned}
\mathbb{E}\, Z^2 &= \mathbb{E}\left(\sum_{a \in \Sigma} f_a h(a)\right) \cdot \left(\sum_{b \in \Sigma} f_b h(b)\right) \\
&= \sum_a f_a^2 \sum_a \mathbb{E}\, h(a)^2 + \sum_{a, b \neq a} f_a f_b\, \mathbb{E}\, h(a) h(b) \\
&= \sum_a f_a^2
\end{aligned}$$

where we use the fact that $\mathbb{E}\, h(a) = 0$ and $\mathbb{E}\, h(a)^2 = 1$ for every a. So the value Z^2 is, in expectation, precisely the F_2 value that we are interested in.

The cancellations in the above calculation may seem a bit magical, so let us try to gain some intuition as to why Z^2 should be, in expectation, the sum-of-squares parameter F_2. Let us first look at a couple of examples as sanity checks: suppose all the elements in the stream are identical; then Z is equal to $+n$ with probability $\frac{1}{2}$ and it is equal to $-n$ with probability $\frac{1}{2}$, so Z^2 is always equal to n^2, which is F_2. If the stream contains two distinct elements, each occurring $n/2$ times, then $Z = 0$ with probability $\frac{1}{2}$ and $Z = \pm n$ with probability $\frac{1}{2}$, and $\mathbb{E}\, Z^2 = \frac{1}{2} n^2$, which is the same as $F_2 = 2 \cdot \left(\frac{n}{2}\right)^2$.

Another way to arrive at the $\mathbb{E}\, Z^2 = F_2$ result is to observe that $\mathbb{E}\, Z = 0$, so $\mathbb{E}\, Z^2 = \mathbf{Var} Z$. This means that we can use the rules to compute variance and see that

$$\mathbf{Var} Z = \mathbf{Var} \sum_a f_a h(a) = \sum_a \mathbf{Var} f_a h(a) = \sum_a f_a^2 \mathbf{Var} h(a) = \sum_a f_a^2$$

Just the fact that our estimator is, on average, equal to the quantity that we want to estimate does not guarantee that we get a good approximation with high probability. The latter is true only we have a small variance. The variance of the random variable Z^2 is

$$\mathbf{Var} Z^2 := \mathbb{E}(Z^2 - (\mathbb{E}\, Z^2))^2 = \mathbb{E}\, Z^4 - (\mathbb{E}\, Z^2)^2$$

We can prove (try to verify it) that

$$\mathbb{E}\, Z^4 = \sum_a f_a^4 + 3 \sum_a \sum_{b \neq a} f_a^2 f_b^2$$

and

$$(\mathbb{E}\, Z^2)^2 = \sum_a f_a^4 + \sum_a \sum_{b \neq a} f_a^2 f_b^2$$

so

$$\mathbf{Var}Z^2 = 2\sum_a \sum_{b\neq a} f_a^2 f_b^2 < 2F_2^2$$

So the average of Z^2 is F_2, and the standard deviation is at most $\sqrt{2}F_2$. Unfortunately this is not yet enough, using the tools that we know (Markov's inequality and Chebyshev's inequality), to argue that we get a constant-factor approximation with good probability. Indeed, in the example in which the stream has two distinct elements each occurring $n/2$ times, Z^2 is 0 with probability .5, so there is a 50% probability of getting an answer that gives no information about F_2.

However, consider the following variant of the algorithm: we pick independently 100 random functions $h_i : \Sigma \to \{-1, 1\}$, for $i = 1, \ldots, 100$, and, for $i = 1, \ldots, 100$, we compute

$$Z_i = \sum_{j=1}^{n} h_i(x_j)$$

which we can do easily by, at every step, computing the 100 hashes of each new item and adding them to the previous values of the Z_i. Our estimate for F_2 will be $\frac{1}{100}\sum_i Z_i^2$.
That is, the pseudocode of the algorithm is

- Pick 100 random functions $h_1, \ldots, h_{100}$, where $h_i : \Sigma \to \{1, -1\}$
- Initialize an integer array Z of size 100 to all zeroes
- while not end of stream
  - read x from the stream
  - for each i in $\{1, \ldots, 100\}$: $Z[i] := Z[i] + h_i(x)$
- return $\frac{1}{100}\sum_{i=1}^{100}(Z[i])^2$

Each Z_i has the same distribution as the random variable Z in the one-function algorithm, so

$$\mathbb{E}\, Z_i^2 = F_2$$
$$\mathbf{Var}Z_i^2 \leq 2F_2^2$$

This means that, if we call A the output of the algorithm, we have

$$\mathbb{E}\, A = \frac{1}{100} \sum_i \mathbb{E}\, Z_i^2 = F_2$$

$$\mathbf{Var} A = \frac{1}{10,000} \sum_i \mathbf{Var} Z_i^2 \leq .02 F_2^2$$

So that

$$\begin{aligned} \Pr[|A - F_2| \geq .5F_2] &= \Pr\left[|A - \mathbb{E}\, A| \geq \sqrt{\frac{.25}{.02}} \cdot \sqrt{.02F_2^2}\right] \\ &\leq \frac{.02}{.25} = .08 \end{aligned}$$

or, in other words, there is at least a 92% probability that

$$.5 \cdot F_2 \leq A \leq 1.5 \cdot F_2$$

The same calculations show that, by picking $\frac{25}{\epsilon^2}$ random functions instead of 100 functions one has at least a 92% probability that

$$(1 - \epsilon) \cdot F_2 \leq A \leq (1 + \epsilon) F_2$$

2 Pseudorandom functions

In the three algorithms that we talked about, the algorithm had to sample and store, respectively, random functions $h : \Sigma \to \{1, \ldots, B\}$, a random function $h : \Sigma \to [0, 1]$, and random functions $h : \Sigma \to \{-1, 1\}$. 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.

If we look back at the analysis of the algorithms, however, we see that we do not need our functions to really be random: we just need them to satisfy certain simple properties that are used in the analysis:

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

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

and the analysis would have worked just as well (with an extra factor of 2 in the approximation) if we instead had

$$\Pr[h(a) = h(b)] \leq \frac{2}{B}$$

- In 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.

Furthermore, for the analysis of the t-th smallest algorithm, we do 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.

- In the analysis of the algorithm for F_2, to compute the expectation of Z^2 we need

$$\mathbb{E}\, h(a)h(b) = (\mathbb{E}\, h(a)) \cdot (\mathbb{E}\, h(b)) = 0$$

for every $a \neq b$. Once more, this just requires the values of h to be pair-wise independent. In the calculation of the variance of Z^2, which we did not fully develop, we also need that for every four distinct labels a, b, c, d, the hash values $h(a), h(b), h(c), h(d)$ be independent.

We see that all the above cases match the following pattern: we have two finite sets Σ and R, and we want to sample a function

$$h : \Sigma \to R$$

such that:

1. For every $a \in \Sigma$, the distribution of $h(a)$ is uniform in R, that is, for every $a \in \Sigma$ and every $r \in R$

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

2. For every two distinct elements $a \neq b$ of Σ, we want $h(a)$ and $h(b)$ to be independently distributed, that is, for every r_1, r_2 in R

$$\Pr[h(a) = r_1 \wedge h(b) = r_2] = \frac{1}{R^2}$$

3. For every four distinct elements a, b, c, d of Σ, we want $h(a), h(b), h(c), h(d)$ to be independent, that is, for every $r_1, r_2, r_3, 4_4$ in R

$$\Pr[h(a) = r_1 \wedge h(b) = r_2 \wedge h(c) = r_3 \wedge h(d) = r_4] = \frac{1}{R^4}$$

For the count-min algorithm for heavy hitters, $R = \{1, \ldots, B\}$ and we want property (2). For the t-th smallest algorithm for the distinct element problem, $R = \{1/N, 2/N, \ldots, 1\}$ and we want properties (1) and (2). For the algorithm for F_2, $R = \{-1, 1\}$ and we want properties (1), (2) and (3).

Note that property (2) implies property (1) and property (3) implies properties (1) and (2). A distribution of hash functions that satisfies property (2) (and (1)) is called *pair-wise independent* and a distribution that satisfies (3) (and (1) and (2)) is called *four-wise independent.*

There are distribution of pair-wise or four-wise independent hash functions such that a function sampled from the distribution can be stored using only $O(\log |\Sigma| + \log |R|)$ bits, provided $|R|$ is a power of a prime number. Here we will sketch a slightly different result, that works for all Σ and R, but introduces a small error.

Say that a distribution of hash functions is *ϵ-close to pairwise independent* if, in property (2), we only guarantee the weaker property that

$$\frac{1}{R^2} - \epsilon \leq \Pr[h(a) = r_1 \wedge h(b) = r_2] \leq \frac{1}{R^2} + \epsilon$$

and define ϵ-close to four-wise independent by similarly modifying property (3).

Theorem 1 *For every $0 < \epsilon \leq 1$ and every two sets Σ, R, a distribution of hash functions*

$$h : \Sigma \to R$$

ϵ-close to pairwise independent can be stored using $O\left(\log |\Sigma| + \log |R| + \log \frac{1}{\epsilon}\right)$ and it can be evaluated in time polynomial in $O\left(\log |\Sigma| + \log |R| + \log \frac{1}{\epsilon}\right)$. The same bounds can be achieved for a distribution of functions ϵ-close to fourwise independent.

In applications, the evaluation simply requires a constant number of addition, multiplication, and mod operations.

Plugging the above theorem into our algorithms, we see that storing the function has the same space complexity as the rest of the algorithm, meaning that, for a fixed approximation parameter ϵ, we can do heavy hitters in $O((\log n) \cdot (\log n + \log |\Sigma|))$ space, number of distinct elements in $O(\log n + \log |\Sigma|)$ space and F_2 is $O(\log n + \log |\Sigma|)$ space.

Now we sketch a proof of the theorem. Let p be a prime number, and suppose that $|\Sigma| = |R| = p$. We will represent both the elements of Σ and the elements of R as the integers in the set $\{0, 1, \ldots, p-1\}$. To sample a hash function, we pick at random x and y in $\{0, \ldots, p-1$, and we define the function $h_{x,y}$ such that

$$h_{x,y}(a) := ax + y \bmod p$$

Then we can use basic linear algebra to show that such a distribution of functions is pair-wise independent. See also section 1.5.2 in the textbook.

If $|R|$ is a prime and $|\Sigma| \leq |R|$, then we use the same construction, by identifying Σ with a subset of $\{0, \ldots, p-1\}$.

Finally, if $|R|$ is not a prime or $|\Sigma| > |R|$, we take a prime $p > \max\left\{|\Sigma|, |R|, \frac{2}{\epsilon}\right\}$. We identify Σ with a subset of $\{0, \ldots, p-1\}$ and we identify R with the set $\{0, \ldots, |R|-1\}$. To generate a random function, we pick at random x, y in $\{0, \ldots, p-1\}$, and we define

$$h_{x,y}(a) := (ax + y \bmod p) \bmod R$$

As before, for every $a \neq b$, the values of $h_{x,y}(a)$ and $h_{x,y}(b)$ are independent. The distribution of $h_{x,y}(a)$ is not uniform, but every possible value v is taken with probability equal to

$$\frac{\{z : 0 \leq z \leq p-1 \wedge z \bmod R = v\}}{p}$$

which is $\frac{1}{p} \cdot \left(\frac{p}{R} \pm 1\right)$, that is between $1/R - 1/p$ and $1/R + 1/p$ and, by our choice of p, at least $1/R - \epsilon/2$ and at most $1/R + \epsilon/2$. This means that, for $a \neq b$, the pair $h(a), h(b)$ takes each possible value in $R \times R$ with probability at least $(1/R - \epsilon/2)^2$, which is at least $1/R^2 - \epsilon$ and at most $(1/R + \epsilon/2)^2$ which is at most $(1/R^2 + \epsilon)$.

The memory use is $2 \log p$. For every n, there is always a prime p such that $n \leq p \leq 2n$, so we can choose $p \leq 2 \cdot \max\{|\Sigma|, |R|, 1/\epsilon\}$ and, with such a choice, $2 \log p = O(\log |\Sigma| + \log |R| + \log 1/\epsilon)$.

The proof for the four-wise independent case is a bit more complicated and we will skip it.

3 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 $O(\epsilon^{-2} \cdot (\log n + \log |\Sigma|))$
- F_2: for every approximation parameter ϵ, given a stream of n elements of Σ, we can compute a number that is, with probability $> 90\%$, between $F_2 \cdot (1 - \epsilon)$ and $F_2 \cdot (1 + \epsilon)$, using space $O(\epsilon^{-2} \cdot (\log n + \log |\Sigma|))$.

4 Memory lower bounds

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, counts the number of distinct elements, or computes F_2, 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 2 *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 3 *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.

The F_2 lower bound is very similar.

Lemma 4 *Suppose that there is a deterministic exact algorithm for computing F_2 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 $o(L)$.

We use the same compression as before: the state of the algorithm after processing

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

where $b_1, \ldots, b_L$ is the string we want to compress. Now, for every i, the value of F_2 for

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

is $n+1$ if $b_i = 1$, because we have $n+1$ distinct elements occurring once each, and it is $n+3$ if $b_i = 0$, because we have $n-1$ elements occurring once and one occurring twice.

Finally,

Lemma 5 *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.