10-806 Foundations of Machine Learning and Data Science

Lecturer: Avrim Blum — Lecture 18: November 9, 2015

---

1 Streaming Algorithms II

Today we continue our discussion of streaming algorithms. Recall the setting: we are observing a sequence (a stream) of m items from a domain of size n, where we think of both m and n as very large. Our goal is to compute some statistic or summary or sketch of this stream using as little space as we can, ideally logarithmic in m and n.

We begin with the problem of estimating the number of distinct elements in the data stream that we got partway through last time.

1.1 Estimating the number of distinct elements

Intuition: if you pick t real numbers uniformly at random in $[0, 1]$, independently, then the expected value of the minimum is $1/(t + 1)$; we won't prove this here but it is a fun puzzle. This suggests the following. Suppose we had a hash function h that mapped each element in the domain to a uniform random number in some range $\{1, \ldots, r\}$, independently. (As we saw earlier, having a truly independent random mapping is too much to hope for, but let's imagine.) If the data stream has t distinct elements and $r \gg t$, then the expected value of $\min\{h(a_i) : a_i \text{ in the stream}\}$ would be approximately $\frac{r}{t+1}$. The key point here is that multiple copies of the same element hash the same way (since h is a function), so $\{h(a_i)\}$ looks like t random numbers between 1 and r. So, from the minimum, we could get an estimate of t. We could then repeat this with multiple hash functions if we want to improve our accuracy.

We'll now implement this idea using a generalization of universal hashing called "2-universal" or "pairwise-independent" hashing. Formally, a set H is a *2-universal* or *pairwise independent* family of hash functions if for any two distinct elements $x \neq x'$ in the domain, and any two elements y, y' in the range,

$$\Pr_{h \in H}[h(x) = y \wedge h(x') = y'] = 1/r^2,$$

where r is the size of the range. Note that our specific scheme using binary matrices also has this property if we remove the all-zeroes element from the domain (since that always hashes to 0).

We're going to focus on getting within a factor of 2 of the correct answer. Note that we're not going to have a hash table now—that would be too big since we'll need r to be larger than t. Instead, we'll just be keeping track of the minimum hash value seen, which takes only $O(\log r)$ bits per hash function. Let's now consider a single hash function chosen at random from our 2-universal family.

Claim 1 *Suppose the data stream $a_1, a_2, \ldots$ has t distinct values. Then, for a 2-universal hash family H,*

$$\Pr_{h \in H}\left[\min_i(h(a_i)) \leq \frac{r}{4t}\right] \leq \frac{1}{4} \quad \textit{and} \quad \Pr_{h \in H}\left[\min_i(h(a_i)) \leq \left\lceil \frac{r}{2t} \right\rceil\right] \geq \frac{3}{8}$$

Proof: First, for any given element a_i, the chance that $h(a_i) \leq \frac{r}{4t}$ is at most $\frac{1}{4t}$; in fact, it's exactly equal to $\frac{1}{4t}$ if $\frac{r}{4t}$ is an integer. So just using the union bound, if there are t distinct elements, the probability that the minimum is $\leq \frac{r}{4t}$ is at most 1/4.

Now, by the same reasoning, the chance that $h(a_i) \leq \lceil \frac{r}{2t} \rceil$ is at least $\frac{1}{2t}$. In fact, to make this analysis cleaner, let's assume $\frac{r}{2t}$ is an integer so the probability is equal to $\frac{1}{2t}$. We now argue that the probability that the minimum is $\leq \frac{r}{2t}$ is at least 3/8. For this we use 2 steps of inclusion-exclusion, called the "Boole-Bonferroni inequalities." These say that for any set of events A_i,

$$\sum_i \Pr(A_i) - \sum_{i<j} \Pr(A_i \cap A_j) \leq \Pr\left[\bigcup_i A_i\right] \leq \sum_i \Pr(A_i).$$

Notice that RHS is the union bound, but now we want to use the LHS. Specifically, since our hash function is 2-universal, for any pair a_i, a_j, $\Pr(h(a_i) \leq \frac{r}{2t} \wedge h(a_j) \leq \frac{r}{2t}) = \frac{1}{4t^2}$. So, applying the inequality we get that $\Pr[\exists i \text{ s.t. } h(a_i) \leq \frac{r}{2t}] \geq \frac{1}{2} - \binom{t}{2}\frac{1}{4t^2} \geq \frac{1}{2} - \frac{1}{8} = \frac{3}{8}$. ■

Now let's perform this with k random hash functions and look at the minima produced by each hash function. We expect at most 1/4 of the minima to be below $\frac{r}{4t}$, and by Hoeffding bounds, for sufficiently large k, with high probability we will have less than 5/16 of the minima below $\frac{r}{4t}$. In the other direction, we expect at most 5/8 of the minima to be above $\lceil \frac{r}{2t} \rceil$ and by Hoeffding bounds, for sufficiently large k, with high probability we will have less than 11/16 of the minima above $\lceil \frac{r}{2t} \rceil$. So, this means that for sufficiently large k, with high probability the $\frac{5}{16} \times 100$th percentile q satisfies:

$$\frac{r}{4t} \;\leq\; q \;\leq\; \lceil \frac{r}{2t} \rceil.$$

So, for $r \gg t$, this means that with high probability we have:

$$t(1 - o(1)) \;\leq\; \frac{r}{2q} \;\leq\; 2t.$$

1.2 Estimating frequency moments

Let $F[i]$ denote the frequency (number of occurences) of element a_i. We're now going to look at the problem of estimating frequency moments, defined as: $f_k = \sum_i F[i]^k$.

Note: f_1 is trivial (it's just m). We can view f_0 as the number of distinct elements if we adopt $0^0 = 0$, and we just looked at that. We will now consider f_k for $k \geq 2$.

Here is an interesting algorithm called the AMS (Alon Matias Szegedy) algorithm:

1. Pick a random element a in the stream and count how many times it appears from that point on (can you see how to modify our procedure for random sampling to do this?). Call this number of occurences r. Also, keep track of the total length of the stream m.

2. Output $X = m(r^k - (r-1)^k)$.

The claim is that this gives an unbiased estimate, i.e., it is correct in expectation. (But the variance may be very large, so one has to repeat many times). Let's see why it has the correct expected value.

Consider some element a_i. By definition it occurs $F[i]$ times. Our algorithm is counting occurences of some element a. The chance we are counting a_i is $F[i]/m$. So, $\mathbf{E}[X] = \sum_i \mathbf{E}[X|a = a_i]F[i]/m$.

Now, what is $E[X|a = a_i]$? It's not $m(F[i]^k - (F[i] - 1)^k)$ because we don't necessarily count a_i from the very first occurence. In fact, given that we are counting occurences of a_i, it is equally likely that our total r will be any number from $1, 2, \ldots, F[i]$. So we get:

$$\mathbf{E}[X|a = a_i] = \frac{1}{F[i]} \sum_{r=1}^{F[i]} m(r^k - (r-1)^k) = \left(\frac{m}{F[i]}\right) F[i]^k.$$

So, overall, $\mathbf{E}[X] = \sum_i F[i]^k = f_k$. This means the expectation is exactly what we want.

Unfortunately, the variance can be very high. If you calculate it out, you get $Var(X) \approx n^{1-1/k}(f_k)^2$.

If you then analyze effect of repetition, it turns out that to get within a constant factor, you need space roughly $O(n^{1-1/k})$ times some log factors in n and m. So it's $o(n)$ but a lot bigger than $\log(n)$. We'll now see a better bound for $k = 2$.

Here is an algorithm for f_2. Called the "tug-of-war" sketch.

Let h be a 4-universal hash function from $\{1, ..., n\}$ to $\{-1, 1\}$.[1]

1. Initialize $X = 0$.
2. When we see some item a, we update: $X = X + h(a)$.
3. Output X^2.

So, in the end, $X = \sum_i F[i]h(a_i)$. We'll then repeat this several times and take the average.

What is $\mathbf{E}[X]$? Clearly this is 0.

What is $\mathbf{E}[X^2]$? This is $\mathbf{E}[(\sum_i F[i]h(a_i))^2] = \mathbf{E}[\sum_i F[i]^2 h(a_i)^2] + 2\mathbf{E}[\sum_{i \neq j} F[i]F[j]h(a_i)h(a_j)] = \sum_i F[i]^2$, where the last equality is coming from the fact that h is pairwise independent, so $\mathbf{E}[h(a_i)h(a_j)] = 0$. So, $\mathbf{E}[X^2] = f_2$, i.e., our output is correct in expectation.

But now we need to look at the variance of our output. This requires looking at $\mathbf{E}[X^4]$ which is why we wanted 4-wise independence.

$\mathbf{E}[X^4] = \mathbf{E}[\sum_{i,j,k,l} F[i]F[j]F[k]F[l]h(a_i)h(a_j)h(a_k)h(a_l)]$.

We can simplify this by noticing that most terms are zero, except for those where $i = j = k = l$ or where the indices pair up, like $i = j$ and $k = l$. There are 3 ways to pair up indices, and then we have to count each pairing twice (equivalently, there are 6 ways to order $\{i, i, j, j\}$) so we get:

$\mathbf{E}[X^4] = 6\sum_{i \neq j} F[i]^2 F[j]^2 + \sum_i F[i]^4 \leq 3(\sum_i F[i]^2)^2 = 3\mathbf{E}[X^2]^2$.

So, $Var(X^2) \leq 2\mathbf{E}[X^2]^2 = 2f_2^2$.

The fact that the variance of our output is comparable with the square of its expectation means that we get get low error by repeating not too many times and taking the average. Specifically, suppose we repeat $r = 6/\epsilon^2$ times and let Z be the average of the estimates X^2. So we have $\mathbf{E}[Z] = f_2$ and now variance has dropped by a factor of r so we have $Var(Z) \leq \mathbf{E}[Z]^2\epsilon^2/3$.

[1] This means that h looks like a truly random function with respect to all 4-tuples (and 3-tuples and pairs and singles). It turns out you can create these from small space too.

Now we can apply Chebyshev's inequality, which says that for any random variable Z and any value b we have $\Pr(|Z - \mathbf{E}[Z]| > b) < \frac{Var(Z)}{b^2}$. Plugging in $b = \epsilon\mathbf{E}[Z]$ we get

$$\Pr(|Z - \mathbf{E}[Z]| > \epsilon\mathbf{E}[Z]) < \frac{Var(Z)}{\epsilon^2\mathbf{E}[Z]^2} < 1/3.$$

So, there's at most a 1/3 chance that our estimate is off by more than an ϵ factor. If we want to reduce this failure probability, one nice thing we can do is repeat this whole process several times and take the *median*. Since these are independent trials, and each has at least a 2/3 probability of success, we can use Hoeffding bounds to say that whp more than half of them will be successful (in the sense of being within an ϵ factor of the correct answer) and so the median will be correct.

As an aside, this idea of hashing with random signs is used as a speedup technique in some learning algorithms—see work of Langford, Smola, and others. The idea is you take your large feature space, and then hash it to a smaller space, but do it with random signs. The idea is that if only a fairly small number of features are highly important, they will hopefully not collide with each other, and the other collisions will tend to cancel out due to the random signs.