10-806 Foundations of Machine Learning and Data Science

Lecturer: Avrim Blum Lecture 17: November 4, 2015

---

1 Streaming Algorithms

We now turn from problems of *prediction* to problems of *summarization*. The setting we will be focusing on is that we have a huge stream of data, like search queries, or packets on a network, and we want to efficiently keep a small "sketch" or summary or set of statistics of this data. The key challenge is that the data stream is extremely large and our memory is small. So, we can't store the whole stream and then do our computation. In fact, we will be especially interested in the case that our memory size is only *logarithmic* in the length of the stream.

Formally, we assume 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 will be to use space logarithmic in m and n. Most of the algorithms we look at will be randomized, and we will assume we have the ability to flip coins of any desired bias.

Notation: We will use $\{a_1, \ldots, a_n\}$ to denote the domain (e.g., if these are search queries, then a_1 might be "machine-learning", a_2 might be "sample-complexity", etc.). We will use F_i to denote the number of occurences (frequency) of element a_i in the data stream. We assume n is known in advance, but m may not be.

1.1 A warmup problem: sampling

As a warmup, let's consider the problem of maintaining an iid sample of k items from the stream, when we don't know the length m of the stream in advance. I.e., at any point in time, we want our sample to consist of k independent draws from the multiset of items observed so far. We will want to do this in space $O(k \log n + \log m)$. Note that it takes space $O(k \log n)$ just to store the sample.

Let's focus on the case $k = 1$, so we just want one random element from the stream. So, the first item comes along and we keep it in memory (which we need to do in case $m = 1$). Now the second item comes along. What do we do? With probability $1/2$ we replace the item in memory with the new one and with probability $1/2$ we keep the old one. Now a third item comes along. We want to replace our memory with the new item with probability $1/3$ and keep the old item with probability $2/3$. In general when the tth item comes along, we put it into our memory with probability $1/t$. We can see why this gives us a uniformly-chosen random element from the stream so far by induction. Clearly the new item ends up in our memory with the correct probability $1/t$, but what about the previous items? Each of them was in our memory with probability $\frac{1}{t-1}$ by induction, so it is in our memory now with probability $\frac{1}{t-1} \cdot \frac{t-1}{t} = \frac{1}{t}$ as desired.

To run this algorithm, we need to keep one item, which requires $O(\log n)$ bits, and the length of the stream so far, which requires $O(\log m)$ bits.

For general values of k, we just run k copies of this independently (but just need to keep one copy of the value of t).

1.2 Maintaining Frequency Counts

Suppose we want to keep track of how many times we've seen each item. We can do this in $O(n \log m)$ space by storing a vector of n counts, one per item, with $\log m$ bits to store each count. Or, we can do this in $O(m \log n)$ space by just storing a list of everything seen, and then just computing the counts when requested. Both of these are too much storage.

Unfortunately, we can't do very much better if we want *exact* information. In fact, suppose all we wanted to know was how many times the *most frequent element* occurred. Even this requires $\Omega(\min(m, n))$ space to keep track of exactly. Intuitively, the difficulty is that we don't know in advace which element is going to be most frequent in the end. Here is the formal reasoning. Let's look at the state of memory after the first $m/2$ items in the stream. Let's just focus on which elements have appeared at least once and which haven't. If $m/2 \geq n$ then there are $2^n - 1$ different subsets that could have appeared. For each subset, arbitrarily fix some sequence containing exactly those elements (each of those elements appearing one or more times). If we have $n-1$ or fewer bits of memory, then there must be some memory state that two different such sequences S_1, S_2 map to. Since each of these sequences corresponds to a different subset, there must be some element that appears in one of them (say S_1) but not the other. Say i is an element that appears $k > 0$ times in S_1 but zero times in S_2. If the next $m/2$ items are all element i, then the correct answer is $m/2 + k$ if the first half was S_1, but is just $m/2$ if the first half was S_2. So we can't tell the difference. If $m/2 < n$, then we can do this with just the first $n' = m/2$ elements and get a lower bound of $m/2$. Either way, we get a lower bound of $\Omega(\min(m, n))$.

Now, what if we allow ourselves an error of ϵm? That is, if element a_i appears F_i times, we want an estimate that is within ϵm of F_i. So, for elements that don't appear very often this isn't very meaningful, but for frequent elements, we want a good estimate. It turns out there are a couple of different interesting ways to do this with very little space. We'll talk about one way, that also has other applications, called the "count-min sketch". The space used will be $O(\frac{1}{\epsilon} \log m)$.

1.3 The Count-Min Sketch

First of all, we are going to maintain estimates $\hat{F}_i$, and the guarantee we'll get is that for each i, our estimate $\hat{F}_i \geq F_i$, and with probability $1 - \delta$, $\hat{F}_i \leq F_i + \epsilon m$.

Here is the idea: We pick $\lg(1/\delta)$ hash functions $h_1, h_2, ..., h_{\lg(1/\delta)}$, each of which is mapping the domain $\{1, 2, \ldots, n\}$ to the range $\{1, 2, ..., r\}$ for $r = 2/\epsilon$. Intuitively, think of them as random mappings from the domain to the range, but formally we will want to pick them from what is called a *universal* hash function family.

Universal Hashing: First of all, the reason we don't really want to choose the h_t's as random mappings is that it would require keeping $\Omega(n)$ bits of information, namely where each item maps to. Instead, a universal hash function family is a set H of hash functions that just satisfies the condition that for any two distinct elements $x \neq x'$ in the domain,

$$\Pr_{h \in H}[h(x) = h(x')] \leq 1/r,$$

where r is the size of the range. The key point is that we can achieve this using a set H that is much smaller than r^n, so that the functions h can be described with many fewer bits. For example, one nice way to do this, if n and r are powers of 2, is as follows. First of all, we will view the domain as the integers $0, 1, \ldots, n-1$. Now, to choose h, we pick a random $\lg(r)$ by $\lg(n)$ matrix of

0s and 1s. Then, viewing the binary representation of x as a column vector, we define $h(x) = hx \pmod 2$, i.e., we multiply the matrix by the vector with all additions done modulo 2. This gives us a $\lg(r)$-bit output, which we view as a binary representation of an element in the range (add 1 if we want the range to be $1, \ldots, r$ instead of $0, \ldots, r-1$). We need to prove that this works, but notice that now h takes only $O((\log n)(\log r))$ bits to describe (and there are even more efficient constructions).

[This is not needed to follow the main argument] Why does this work? The easiest way to see it is to think of hx as using x to index a subset of columns of h and then adding those columns up mod 2. If $x \neq x'$, then there must be at least one index i where $x[i] \neq x'[i]$, and let's say $x[i] = 0$ and $x'[i] = 1$. Fixing x and x', over a random choice of h, we want to argue that the chance of $h(x) = h(x')$ is at most $1/r$. Imagine filling h randomly with 0's and 1's but filling in the ith column last. Just before the ith column is filled, $h(x)$ is now determined. But all of the $2^{\lg(r)}$ ways of filling in the ith column produce a *different* value for $h(x')$. That's because if $a + b = a + b'$ then $b = b'$ (where b, b' represent two different possible ith columns and $a = h(x')$ if you fill in the ith column with all zeroes). So, only one of those r ways can collide with $h(x)$.

Back to the Count-Min Sketch: So, we have our $\lg(1/\delta)$ hash functions. Now what we'll do is keep a $\lg(1/\delta)$ by r matrix of counts, called "count", with all entries initialized to 0. When the item a_i arrives, we will increment $count[j, h_j(a_i)]$ for each $j = 1, 2, \ldots, \lg(1/\delta)$. Think of each row of this matrix as a hash table but where we don't resolve collisions and instead just increment the counters. So, notice that $count[j, h_j(a_i)]$ is always greater than or equal to the true number of occurences of a_i. Our estimate $\hat{F}_i$ will be $\min_j count[j, h_j(a_i)]$.

To analyze this, we need to analyze collisions. Fix some element a_i and some hash function j. The expected number of collisions of other elements into $h_j(a_i)$ is at most m/r since each other element has at most a $1/r$ probability of colliding with a_i, over the selection of h_j. Here we are using linearity of expectation along with the fact that h_j was selected from a universal hash function family. So the expected number of collisions with a_i is at most $\epsilon m/2$. By Markov's inequality, this means there is at least a $1/2$ chance that the number of collisions with a_i is at most ϵm. Now, we don't necessarily get tail bounds for an individual hash function, but since we chose the hash functions independently from the universal hash family, the chance that *all* of them have more than ϵm collisions is at most $(1/2)^{\lg(1/\delta)} = \delta$. So, with probability $\geq 1 - \delta$ we have $\hat{F}_i \leq F_i + \epsilon m$, as desired.

Another nice application is this also works with inserts/deletes (each observation is an element a_i along with a positive or negative change in its "account". Define m now as the sum of all accounts at the end of the process. All the rest of the analysis still holds (except space usage depends on the log of the maximum count value over time, not just the final count value).

1.4 Estimating the number of distinct elements

Another quantity we might want to estimate in our data stream is the number of *distinct* elements seen. Note that the above procedure doesn't really help with this.

Here is an idea for how we can do this. First, some intuition: 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 above, having a truly independent random mapping is too much to hope for, but let's imagine.) If the data stream has t distinct elements, then the expected value of $\min_i h(a_i)$ would be approximately $\frac{r}{t+1}$; here, think of r as much larger than t. 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,*

$$\Pr_{h \in H}\left[\min_i(h(a_i)) \leq \frac{r}{4t}\right] \leq \frac{1}{4} \quad \text{and} \quad \Pr_{h \in H}\left[\min_i(h(a_i)) \geq \left\lceil \frac{r}{2t} \right\rceil\right] \leq \frac{5}{8}$$

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:

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