Lecture 6: Data Streaming

January 24, 2019

Lecturer: Eric Vigoda *Scribes: Caitlin Beecham, Qiaomei Li*

6.1 Data Streaming

General idea: We want to count d, the number of distinct values that appear in our stream S. However, storing the desired frequency vector might take a lot of space. So instead of storing it, we hash the values, k, that appear in the stream one by one (Side note: hashing these values takes the arbitrary distribution of values which appears in S and makes the hashed output roughly uniform over the set $\{0, 1, \ldots, p-1\}$). Then, one by one we add $(h(k), zeros(h(k))) = (h(k),$[the number of trailing zeros in h(k)] when written in binary]) to a set (NOT a multiset) B. The idea is that if we know the number of times we have a hashed value $h(k)$ with more than some number of zeros, this number gives us an estimate on the number of distinct values that appear in the stream. In particular, say we have a hashed value $h(k)$ written in binary. Since the hashed distribution is uniform, we know that the probability that $h(k)$ has at least l trailing zeros is $= \frac{1}{2^l}$. So, the probability that an element of B has $\geq l$ zeros is about $\frac{d}{2^l}$ (I say about because there could be hash collisions). Now, as this algorithm goes on, we actually make sure to only keep elements of B in B for which the number of trailing zeros is above this threshold. So, at the end of the algorithm, B consists of only the elements with enough trailing zeros. The expected number of these is $E[|B|] = \frac{d}{2^l}$. Thus, our algorithm outputting $|B|2^l$ should be a reasonably good estimate of d.
Note: the answer to the question asked during class about whether we could just store $zeros(h(k))$ instead of $(h(k), zeros(h(k)))$ is that we can't. The crux of what makes this algorithm works is that we are storing the elements of B as a set and not a multiset, which is what gives us some kind of relation to the number of distinct elements d (because we're not keeping elements with multiplicity). So, in particular, what allows us to know that we are only keeping one copy of each value (well, really we're keeping the hashed value, but distinct values give distinct hash values) is that we actually know that value $h(k)$ because it is stored. If we only stored $zeros(h(k))$, we would end up counting many distinct hash values as one thing because they have the same number of trailing zeros. So, basically the whole reason we have a connection between d and $|B|$ is that we are storing all hash values that appear and storing them exactly once (until they get thrown out). If we only kept track of $zeros(h(k))$ instead of $(h(k), zeros(h(k)))$ we would lose that connection.

Let the list $S = \{S_1, ..., S_m\}$ s.t. m is huge and $f = (f_1, ..., f_n)$, where $f_i = |\{j : 1 \leq j \leq m, s_j = i\}|$. Our goal is to compute $d = F_0 - |\{i : f_i > 0\}|$=the number of distinct items. From last class we found $\hat{d}$ where $Pr(\frac{\hat{d}}{3} \geq d \geq 3\hat{d}) \geq 0.04$. Next, we want to boost it to be $\geq 1 - \delta$ by finding the median of $O(log(\frac{1}{\delta})$ trials.
We show $\forall \epsilon > 0, \delta > 0$, $Pr(\hat{d}(1-\epsilon) \geq d \geq \hat{d}(1+\epsilon)) \geq 1 - \epsilon$. Given $h : [n] \rightarrow [n]$, $Pr(zeros(h(k)) \geq l) = 2^{-l}$. Recall from last class, the algorithm that finds $\max\{l : zeros(h(k)), k \in S\}$ outputs $2^{l+1/2}$. Consider the problem of finding the number of k such that $zeros(h(k)) \geq l$, we expect $\frac{d}{2^l} = |B|$, and the corresponding algorithm outputs $|B|2^l$.

choose a random pairwise independent hash function $h : \{[1, ..., n\} \to \{[1, ..., n\}$;
find prime p s.t. $n \leq p \leq 2n$; choose a,b independently and uniformly from $\{0, 1, ..., p-1\}$ and define the hash function $h(i) = a + bi \bmod p$;
set the counter $z = 0$, and $B = \emptyset$;
goes through data stream S, at k: hash it, look at $zeros(h(k))$
if *zeros(h(k))*$\geq z$ **then**
    $B = B \cup (h(k), zeros(h(k)))$;
    while $|B| > 100/\epsilon^2$ **do**
        $z = z + 1$;
        remove all (α, β) *from* B *where* $\beta < z$
output $|B|2^Z$

6.1.1 The algorithm

For $k \in \{1, ..., n\}$ and integer $l \geq 1$,

$$X_{l,k} = \begin{cases} 1, & \text{if } zeros(h(k)) \geq l \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

Let $Y_l = \sum_{k: f_k > 0} X_{l,k}$, output $\hat{d} = Y_z 2^z$.

$$E[Y_l] = \sum_{k:k>0} E[X_{l,k}] = \frac{d}{2^l} \tag{6.2}$$

$$Var(Y_l) = Var(\sum_{k:k>0} X_{l,k}) \tag{6.3}$$

$$= \sum_{k:k>0} Var(X_{l,k}) \tag{6.4}$$

$$\leq \sum_{k:k>0} E[X_{l,k}^2] \tag{6.5}$$

$$= \sum_{k:k>0} E[X_{l,k}] = \frac{d}{2^l} \tag{6.6}$$

We want $(1 - \epsilon)\hat{d} \leq d \leq (1 + \epsilon)\hat{d}$

Our algorithm fails if $|\hat{d} - d| > \epsilon d$. Note that $|\hat{d} - d| = |y_z 2^z - d| > \epsilon d$ if and only if $|y_z - \frac{d}{2^z}| > \frac{\epsilon d}{2^z}$. Intuitively, we break this into two cases: if $z <$ [some threshold] we bound using Chebyshev and if $z \geq$ [that threshold] we use Markov's Inequality to say that having such a large z is unlikely to happen.
Now, $Pr(|Y_z - E[Y_z]| > \frac{\epsilon d}{2^z}) \leq \frac{\mathrm{Var}(Y_z)}{(\frac{\epsilon d}{2^z})^2} \leq \frac{1}{\epsilon^2 (\frac{d}{2^z})}$.
So, $Pr(\text{FAILURE}) = Pr(|Y_z - E[Y_z]| > \frac{\epsilon d}{2^z}) = \sum_{r=0}^{\log(n)} Pr(|Y_r - E[Y_r]| > \frac{\epsilon d}{2^r} \text{ AND } (z = r))$. We break this summand into two sums. Note that we use the following fact: $Pr(A \text{ AND } B) \leq Pr(A)$ and also

$Pr(A \text{ AND } B) \leq Pr(B)$.

$$\sum_{r=0}^{\log(n)} Pr(|Y_r - E[Y_r]| > \frac{\epsilon d}{2^r} \text{ AND } (z = r)) = \sum_{r=0}^{s-1} Pr(|Y_r - E[Y_r]| > \frac{\epsilon d}{2^r} \text{ AND } (z = r)) + \tag{6.7}$$

$$\sum_{r=s}^{\log(n)} Pr(|Y_r - E[Y_r]| > \frac{\epsilon d}{2^r} \text{ AND } (z = r)) \tag{6.8}$$

$$\leq \sum_{r=0}^{s-1} Pr(|Y_r - E[Y_r]| > \frac{\epsilon d}{2^r}) + \tag{6.9}$$

$$\sum_{r=s}^{\log(n)} Pr(z = r) \tag{6.10}$$

$$= \sum_{r=0}^{s-1} \frac{2^r}{\epsilon^2 d} + Pr(z \geq s) \tag{6.11}$$

$$= \frac{1}{\epsilon^2 d}(\sum_{r=0}^{s-1} 2^r) + Pr(Y_{s-1} > \frac{1000}{\epsilon^2}) \tag{6.12}$$

$$\leq \frac{2^s}{\epsilon^2 d} + \frac{\epsilon^2 E[Y_{s-1}]}{1000} \tag{6.13}$$

$$= \frac{2^s}{\epsilon^2 d} + \frac{d\epsilon^2}{1000 * 2^{s-1}} \tag{6.14}$$

Choose $s = \max\{n \in \mathbb{N} | \frac{d}{2^s} < \frac{24}{\epsilon^2}\} \leq \frac{2^s}{\epsilon^2 d} + \frac{2\epsilon^2}{1000}\frac{24}{\epsilon^2} = \frac{2^s}{\epsilon^2 d} + \frac{48}{1000} \leq \frac{2^s}{\epsilon^2 d} + \frac{1}{12} \leq \frac{1}{12} + \frac{1}{12} = \frac{1}{6}$. Now, $\frac{12}{\epsilon^2} \leq \frac{d}{2^s}$ implies that $2^s \leq \frac{d\epsilon^2}{12}$ so that $s = O(\log_2(c' d\epsilon^2))$. What space does this algorithm use? Well, each $h(k)$ uses $O(log(n))$ bits. Also, zeros($h(k)$) uses $O(log(n))$ bits. So, overall, this algorithm uses $O(log(n)) * O(\frac{1}{\epsilon^2})$ bits to store B and also $O(log(log(n)))$ bits to store z and finally $O(log(n))$ bits to store a and b. In total, it uses $O(log(n)) * O(\frac{1}{\epsilon^2}) + O(log(log(n))) + O(log(n)) = O(log(n)) * O(\frac{1}{\epsilon^2})$ bits. So, what if we want to do better than $\frac{1}{\epsilon^2}$? Really, we should take $O(\log(\frac{1}{\epsilon^2}))$ to remember each item in B, which means that we should make another hash function. We still use the original hash function $h : [n] \rightarrow [n]$, as well as a new hash function, $g : [n] \rightarrow [\frac{10^6}{\epsilon^2}]$ with some constant probability collisions.