CS 15-851: Algorithms for Big Data Spring 2024

Lecture 9 — Mar 21

Prof. David Woodruff *Scribe: Trung Tran*

1 Heavy Hitter Guarantees

Recall from the past lecture, the goal of Heavy Hitter problem is to find big coordinates in the underlying vector x. Depending on what we mean by "big coordinates", we can define different guarantees for Heavy Hitter problem.

- ℓ_1**-guarantee.** We want to output all indices j for which $|x_j| \geq \phi \|x\|_1$. However, this strict constraint would lead to an $\Omega(n)$ lower bound. Therefore, we allow approximation in the sense that the outputted set might have false positive but should not contain any indices j such that $|x_j| \leq (\phi - \varepsilon) \|x\|_1$ for some control error ε.
- ℓ_2**-guarantee.** Similarly, we relax the problem and ask for a set containing ALL the indices j for which $|x_j|^2 \geq \phi \|x\|_2^2$ and not containing ANY j such that $|x_j|^2 \leq (\phi - \varepsilon) \|x\|_2^2$ for some ε.

Note that, ℓ_2-guarantee can be much stronger than ℓ_1-guarantee. Take $x = (\sqrt{n}, 1, 1, ..., 1)$ as an example, where x is an n-dimensional vector with all entries but the first coordinate being 1. It is not hard to see the item corresponding to the first coordinate is $\frac{1}{\sqrt{n}}$-ℓ_1 heavy hitter. This could be a problem as there might be $\sqrt{n}$ items that are $\frac{1}{\sqrt{n}}$-ℓ_1 heavy hitter, which potentially need a lot of memory to recover. On the other hand, we only require $\phi = \frac{1}{2}$ for the heavy item from ℓ_2 guarantee perspective. More formally, if an index j is ϕ-ℓ_1 heavy hitter, then it is also ϕ^2-ℓ_2 heavy hitter:

$$|x_j| \geq \phi \|x\|_1 \Rightarrow x_j^2 \geq \phi^2 \|x\|_2$$

One could ask if we can go beyond ℓ_2 guarantee and get ℓ_p guarantee with $p > 2$. Unfortunately, as we will see soon, the memory needed in that case grows polynomially in n.

2 Building intuition through special cases

2.1 Case study 1

Suppose we are promised at the end of a stream, we have some entry $x_i = n$ and $x_j \in \{0, 1\}, \forall j \neq i$. How would we find the identity of i?
Interestingly, there is a deterministic way to identify i using $\log^2 n$ memory. Consider $\log n$ buckets where we hash a new update (i, Δ) to every bucket j such that the j-th bit in the binary representation of i is 1. At the end of the stream,

- If bucket i contains the heavy item, the total weight of bucket i is at least n.
- Otherwise, the total weight of bucket i is at most $n-1$.

The gap between two cases helps us to read off every bucket having the heavy item whose binary representation is completely defined by this information.
Since we need $\log n$ bit to store a counter for every bucket, the total amount of memory we need is $\log^2 n$.

2.2 Case study 2

Now consider a slightly different setup in which we are promised $x_i = 100\sqrt{n \log n}$ and $x_j \in \{0, 1\}, \forall j \neq i$. It is not hard to see a direct application of the previous algorithm does not work as the heavy item is not big enough. However, we could make it work with a small modification. Divide the set of all items $(1, 2, ..., n)$ into chunks of size $\sqrt{n \log n}$, each of which is handled by $\log n$ counters as the original algorithm. The total amount of space we use by following this scheme is the number of chunks $\times \log^2 n = \sqrt{n} \log^{1.5} n$. This is an improvement over the trivial $\Omega(n)$ memory, but we would like to do much better (the dependency is polynomial in $\log n$ instead of n).
The key idea is to multiply each item with a random sign before hashing it to buckets as usual. Using an additive Chernoff bound, the noise (i.e. the total weight of light items) in each bucket is bounded by $\sqrt{n \log n}$ with high probability. Thus, for buckets that do not contain the heavy item, the corresponding counters are in the range $\pm\sqrt{n \log n}$, while the counters of the others are at least $100\sqrt{n \log n} - \sqrt{n \log n} = 99\sqrt{n \log n}$ in absolute value. This gap once again helps us to read off the binary representation of the heavy item.
So far, we have played around with some special cases to gain intuition. In the next part, we are going to remove assumption $x_i = n$ or $100\sqrt{n \log n}$ and $x_j = 0$ or 1, and get an algorithm that works in general case using CountSketch.

3 CountSketch achieves ℓ_2-guarantee

The previous examples motivate the idea of hashing into buckets and taking sums of weights that are scaled by random signs. This sounds like a job for CountSketch. More specifically, we assign each coordinate i a random sign $\sigma_i \in \{-1, 1\}$ and randomly partition all coordinates into B buckets using a hash function $h : [n] \to [B]$. Let

$$c_j = \sum_{h(i)=j} \sigma_i x_i$$

be the counter for the j-th bucket. We can then estimate x_i as $\sigma_i x_{h(i)}$. To demonstrate why this is a reasonable estimation, we will first show the above estimator is unbiased. Indeed,

$$\begin{aligned}
\mathbf{E}\left(\sigma_i c_{h(i)}\right) &= \mathbf{E}\left(\sigma_i \sum_{i':h(i')=h(i)} \sigma_{i'} x_{i'}\right) \\
&= \mathbf{E}\left(\sigma_i^2 x_i + \sum_{i':h(i')=h(i),i'\neq i} \sigma_i \sigma_{i'} x_{i'}\right) \\
&= x_i + \mathbf{E}\left(\sum_{i':h(i')=h(i),i'\neq i} \sigma_i \sigma_{i'} x_{i'}\right) \\
&= x_i
\end{aligned}$$

To show the concentration of the estimator around its expectation, we will use the second moment method. The amount of "noise" in the i-th bucket contaminating our estimate is

$$\sigma_i \sum_{i':h(i')=h(i),i'\neq i} \sigma_{i'} x_{i'}$$

The second moment of the noise in bucket i thus is

$$\mathbf{E}\left(\sum_{i':h(i')=h(i),i'\neq i} \sigma_i \sigma_{i'} x_{i'}\right) = \frac{\|x\|_2^2 - x_i^2}{B} \leq \frac{\|x\|_2^2}{B}$$

By Chebyshev inequality, the noise in a bucket is $O\left(\frac{\|x\|_2}{\sqrt{B}}\right)$ in magnitude with constant probability. To boost the success probability, we can apply the median trick that comes up all over the place in our class: independently repeat this hashing scheme $O(\log n)$ times, and output the median of the estimates across the $O(\log n)$ repetitions. Since all of the repetitions are independent, this ensures that

$$\sigma_i c_{h(i)} = x_i \pm O\left(\frac{\|x\|_2}{\sqrt{B}}\right)$$

with failure probability $\frac{1}{\text{poly}(n)}$. By union bound, we can approximate every x_i simultaneously up to additive error $O\left(\frac{\|x\|_2}{\sqrt{B}}\right)$ with high probability. Lastly, we can set $B = O\left(\frac{1}{\varepsilon^2}\right)$ (note that $B = O\left(\frac{1}{\varepsilon}\right)$ is not strong enough) to approximate x_i^2 up to additive error $\frac{\varepsilon \|x\|_2^2}{2}$, which is good enough to distinguish 2 cases $x_i^2 \geq \phi \|x\|_2^2$ and $x_i^2 \leq (\phi - \varepsilon)\|x\|_2^2$.

4 Tail guarantee

We just proved that CountSketch approximates every x_i simultaneously up to additive error $O\left(\frac{\|x\|_2}{\sqrt{B}}\right)$, but sometimes this is not good enough. For example, consider a vector

$$x = [\text{poly}(n), n, 1, 1, ..., 1]$$

where the first item is super large, the second item is less large, and all the other items are 1. The norm of x will be dominated by the first item, so that we will not get a good approximation for the second item, which in some cases, we may still be interested in it because it is still relatively large compared to the rest of the coordinates. It would be nice if we could get an estimate for the next heaviest hitter after removing the largest entry, and and do this in one pass in a stream. In fact, the algorithm presented in the previous section actually does give us this tail guarantee.

Tail guarantee. CountSketch approximates every x_i simultaneously up to additive error $O\left(\frac{\left\|x_{B/4}\right\|_2}{\sqrt{B}}\right)$, where $x_{B/4}$ is x after zeroing out its top $B/4$ coordinates in magnitude.

Proof. By union bound, in each repetition, the top $B/4$ coordinates of x do not land in the same hash bucket as x_i with probability at least $\frac{3}{4}$. Condition on x_i not colliding with any one in the top $B/4$ heaviest items, the same old analysis tells us the noise in the i-th bucket is $O\left(\frac{\left\|x_{B/4}\right\|_2}{\sqrt{B}}\right)$ in magnitude with constant probability. To boost the success probability and get a bound simultaneously for all i, we can just apply the median trick and do the union bound over all indices i.

Further, we only needed pairwise independence of the hash functions for this proof (to compute the second moment of the noise), which means the hash functions can still be stored efficiently.

It is also interesting to note that, if x is $B/4$-sparse this result implies that all x_i are estimated exactly.

5 Finding heavy hitters quickly

The above algorithm produces estimates for all coordinates x_i using the optimal space efficiency. However, it takes $O(n)$ time to decode Sx and find the heavy hitter coordinates as we will need to loop through the entire resulting vector and test whether each coordinate is above the threshold. Is there any way to achieve faster decoding step with slight memory trade-off? We can accelerate this using a binary-tree-like data structure, in which we partition all entries of x into $\frac{C}{\varepsilon}$ blocks where C is some constant. Compute the sum of coordinates in each block as the new vector to sketch (We will assume the vector x is non-negative). If there is a heavy hitter in a block, then the block itself must be a heavy hitter for the new vector. Thus, taking all heavy hitter blocks at this level gives us a superset of the actual heavy hitter coordinates.

At the next level, we split each block from the previous level into two blocks. Thus, we will have $\frac{2C}{\varepsilon}$ blocks at the second level. Now, once we decode the first level and find the heavy hitter blocks at the first level, we do not need to decode all blocks from the second level. Specifically, at most $\frac{1}{\varepsilon}$ blocks are heavy in the first level, and we only decode the second level blocks that come from these heavy first level blocks. The process is similar to searching through a binary tree. It is the case that there will be at most $\frac{1}{\varepsilon}$ heavy blocks at each level, requiring $O\left(\frac{1}{\varepsilon}\right)$ time to decode, for each of the $\log n$ levels. This new algorithm takes extra $\log^4 n$ memory due to $\log n$ storage of CountSketch in this binary tree.

6 ℓ_1 guarantee

As shown in section 1, the ℓ_2-guarantee implies the ℓ_1-guarantee. So, why should we care about ℓ_1-guarantee? One nice thing about ℓ_1-guarantee is that it can be solved deterministically! To do so, we consider a so-called ε-incoherence matrix S defined as follows:

Incoherence matrix. An $s \times n$ matrix S is ε-incoherent if all columns S_i have norm $\|S_i\|_2 = 1$ and any pair of columns has dot product $|\langle S_i, S_j \rangle| < \varepsilon$.

It is not clear why such matrices even exist. We will present a construction of such matrices later. For now, let us suppose they exist. Furthermore, entries in S can be specified with $O(\log n)$ bits of space. We can maintain Sx in a stream using $O(s \log n)$ bits of space and outputs an estimate $\widehat{x_i} = \langle Sx, S_i \rangle$.

To see how this works, note that by ε-incoherence property we have:

$$\begin{aligned}
\widehat{x_i} &= \langle Sx, S_i \rangle \\
&= \sum_{j=1}^{n} x_j \langle S_j, S_i \rangle \\
&= x_i \langle S_i, S_i \rangle + \sum_{j \neq i}^{n} x_j \langle S_j, S_i \rangle \\
&= x_i + \sum_{j \neq i}^{n} x_j \langle S_j, S_i \rangle \\
&= x_i \pm \varepsilon \|x\|_1
\end{aligned}$$

Similar to the ℓ_2 heavy hitter problem, by picking $\varepsilon = \frac{\varepsilon}{2}$, we can distinguish 2 cases $|x_j| \geq \phi \|x\|_1$ and $|x_j| \leq (\phi - \varepsilon) \|x\|_1$, thus solving ℓ_1 heavy hitter.

We now turn to constructing ε-incoherent matrices. Consider a prime $q = \Theta\left(\frac{\log n}{\varepsilon}\right)$ (the existence of such a primes follows Bertrand-Chebyshev theorem saying there is a prime number between n and $2n$ for all $n > 1$) and $d = \varepsilon q = \Theta(\log n)$. Let $P_1, P_2, ..., P_n$ be n distinct nonzero polynomials

$$P_i = \sum_{j=0}^{d-1} a_{i,j} x^j \mod q$$

for each of which there are $q^d - 1$ possibilities. Choosing q such that $q^d - 1 > n$, associate each polynomial to a column of S and set the number of rows to be $s = q^2$. Partition S into q chunks of q rows and populate S by setting the $P_i(j)$-th entry of the i-th column in the j-th chunk to have entry $\frac{1}{\sqrt{q}}$, with all other entries being 0. Each column S_i thus has $\|S_i\|_2 = 1$, and the dot product between S_i and S_j is the number of entries k such that $P_i(k) = P_j(k)$ divided by q. Since $P_i - P_j$ is a polynomial of degree at most $d - 1$, it has at most $d - 1$ zeros and so the dot product is bounded by $\frac{d-1}{q}$. Since $d = \varepsilon q$ we have that each dot product is bounded by ε. Thus we can achieve an ε-incoherent matrix with $O\left(\frac{\log^2 n}{\varepsilon^2}\right)$ rows.