CS 15-859: Algorithms for Big Data Fall 2020

Lecture 7 — October 22, 2020

Prof. David Woodruff *Scribe: Michael Zhang*

1 Quick Recap of ℓ_1-Regression

So far, we have discussed most of the steps of the fast algorithm of computing the ℓ_1-regression problem using a sketching matrix of i.i.d. Cauchy random variables. What is yet to be discussed is how to speedup the bottleneck operation of this algorithm - computing $R \times A$ where R is the $(d \log d) \times n$ matrix of i.i.d. Cauchy random variables. Since R is dense, computing RA will take $O(n \times \text{poly}(d))$ times. The other steps in the algorithm are much faster since they are in a much lower dimension of poly(d).

We can speed up this step by choosing R to be the product of a countsketch matrix and a diagonal matrix consists of i.i.d. Cauchy random variables (i.e. $R = SC$ where S is a $d/\varepsilon \times (d \log d)$ countsketch matrix and C is a diagonal matrix with i.i.d. Cauchy random variables).

With R chosen as above, we will get a slightly looser guarantee on $\|RAx\|_1$:

$$\frac{1}{d^2 \log d}\|Ax\|_1 \leq \|RAx\|_1 \leq O(d \log d)\|Ax\|_1$$

However, we can achieve a much faster run time with $O(\text{nnz}(A) + \text{poly}(d/\varepsilon))$.

We can even further improve the algorithm to with $R = SE$ where S is a $d/\varepsilon \times (d \log d)$ countsketch matrix and E is a diagonal matrix of the reciprocals of exponential random variables. This achieves a better error bound for $\|RAx\|_1$:

$$\frac{1}{d^{0.5}\text{poly}(\log(nd))}\|Ax\|_1 \leq \|RAx\|_1 \leq O(d \log d)\|Ax\|_1$$

Recently, there is an even better bound discovered by Wang and Woodruff [1] in 2019, which gives us an error bound of

$$\Omega(1) \leq \|RAx\|_1 \leq O(d \log d)\|Ax\|_1$$

with similar complexity.

2 A Fun Fact About Cauchy Random Variables

Suppose we have i.i.d. copies of $R_1, \ldots, R_n$ of a random variable with 0 mean and σ^2 variance. Intuitively, when n gets very large, the mean of these random variables approaches 0, while the variance is also decreasing. In fact, this is an important theorem in statistics:

Central Limit Theorem: The distribution of $\frac{\sum_i R_i}{n}$ approaches the normal random variable of $N(0, \sigma^2/n)$ when n approaches infinity.

However, there is a counter example - if $R_1, \ldots, R_n$ are Cauchy random variables, the above theorem does not hold. Let c be a Cauchy random variable,

$$\frac{\sum_i R_i}{n} \sim \frac{n \cdot c}{n} \sim c$$

The mean is still 0 but the variance does not decrease even if we have a large number of R's. In other words, CLT does not apply to Cauchy random variables.

3 Introduction to Streaming

In the next part of the course, we shift our focus to another important data model - Streaming. In a lot of real world applications, we cannot afford the space to store all the data involved in a problem, therefore the "one pass" streaming model is of great interest in the study of big data algorithms. Let's begin by introducing a model of Streaming problems.

3.1 Turnstile Streaming Model

- There is an underlying n-dimensional vector x initialized to 0^n
- There is a long stream of updates $x_i \leftarrow x_i + \Delta_j$ for Δ_j in $\{-M, -M+1, \ldots, M-1, M\}^n$ where $M \in \text{poly}(n)$.
- Throughout the stream, x is promised to be in $\{-M, -M+1, \ldots, M-1, M\}^n$ (i.e. the entries of x could be stored with $O(\log n)$ bits).
- Output an approximation to $f(x)$ with high probability over our coin tosses (i.e. we may choose to use a randomized algorithm)
- The goal is to use as little space (in bits) as possible.

There are a lot of real world applications to this streaming model. For example, if we are interested in the packets over the Internet, stock transactions, weather data, and genome sequences, this model becomes very useful since we normally cannot afford to store all the input data into our memory.

3.2 Some Example Problems

Now we discuss some example streaming problems and attempt to solve these problems using the techniques we discussed in this course before.

Example 1. Testing if $x = 0^n$ with probability at least 9/10.

Observe that a trivial deterministic algorithm is to store every entry using $O(n \log n)$ space. First, let's show that there is no deterministic algorithm with any improvement in space.

Claim 1. Any deterministic algorithm requires $\Omega(n \log n)$ bits of space.

Proof of Claim 1: AFSOC we have such a deterministic algorithm that uses less than $\Omega(n \log n)$ bits of space. Consider a stream where the first half corresponds to a vector $a \in \{0, 1, 2, \ldots, \text{poly}(n)\}^n$. Let $S(a)$ be the state of the algorithm after reading the first half of the stream. By Pigeon-hole Principal, if $\|S(a)\| = o(n \log n)$, there exists a state $a' \neq a$ such that $S(a) = S(a')$ because $2^{o(n \log n)} < n^{\text{poly}(n)}$. Suppose the second half of the stream corresponds to $b \in \{0, -1, -2, \ldots, -\text{poly}(n)\}^n$ which essentially "undo" the first half. Now $S(a + b)$ and $S(a' + b)$ must be different since one is 0^n and one is not. Therefore we have a contradiction.

Claim 2. There is a randomized algorithm with $O(\log n)$ bits of space.

Algorithm: Le S be a $1/\varepsilon^2 \times n$ countsketch matrix. We store and update the entries of Sx along the stream.

Proof: Since each column in S has only 1 non-zero entry and the map is linear, $Sx \leftarrow Sx + \Delta_j S e_i$ where i is the index of x being updated. By the JL-moment property, with probability at least 9/10,

$$\|Sx\|_2^2 = O(1 + \varepsilon)\|x\|_2^2$$

Therefore, if $x = 0$, then with probability 1 we have $Sx = 0$. When $x \neq 0$, with probability at least 9/10, $Sx \neq 0$. We get a one-sided-error algorithm with probability of success at least 9/10. If we set $\varepsilon = \frac{1}{2}$, the overall space needed is only $O(\log n)$ bits. Note that we can store S as a 4-universal hash function and 2-universal sign function that defines S in $O(\log n)$ bits as well.

Example 2. Recovering a k-sparse vector

Suppose we are promised that x has at most k non-zero entries at the end of the stream, and we want to precisely recover all those k entries at the end.

Claim 3. There exists a deterministic algorithm for this problem with $O(k \log n)$ bits of space.

Algorithm: Suppose A is an $s \times n$ matrix such that any 2k columns are linearly independent. Then we maintain Ax in the stream for each update. At the end we can recover x from Ax.

Proof: First we show that it is indeed possible to recover the subset of k non-zero entries and their values from Ax.

AFSOC $\exists y \neq x$ where $Ax = Ay$. Then $A(x - y) = 0$. Since x and y each only has at most k non-zero entries, $x - y$ has at most 2k non-zero entries. Therefore, $A(x - y)$ is a linear combination of 2k columns of A. This contradicts with the assumption that any 2k columns of A are linearly independent. Therefore x is indeed unique and this algorithm is deterministic.

But what is this matrix A? It turns out it is very similar to the $2k \times n$ Vandermonde matrix defined as follows:

$$A_{i,j} = j^{i-1}$$

This matrix looks like $\begin{bmatrix} 1, 1, 1, \ldots \\ 1, 2, 3, \ldots \\ 1, 4, 9, \ldots \\ 1, 8, 27, \ldots \end{bmatrix}$, where the determinant of any $2k \times 2k$ submatrix of A with set of columns equal to $\{i_1, \ldots, i_k\}$ is $\Pi_j i_j \Pi_{j' < j}(i_{j'} - i_j) \neq 0$, so any 2k columns are linearly independent.

Note that the entries in A is exponentially increasing and $\log(n^{2k}) = O(k \log n)$ might be too big to store, so applying A directly to x might not work. To resolve this problem, we pick a large

enough prime $p > 2M$ and $p \in \text{poly}(n)$. Check that the condition of A above still holds because if $A(x-y) \equiv 0 \mod p$, we know that either $x - y \equiv 0 \mod p$ or $A \equiv 0 \mod p$. However, since $-2M \leq x - y \leq 2M$, $x - y \neq 0 \mod p$ and $\Pi_j i_j \Pi_{j'<j}(i_{j'} - i_j) \neq 0 \mod p$. Therefore the proof above still holds, and any 2k columns of A are linearly independent.

We do not need to explicitly store A since A is defined by a function. The overall space complexity of this algorithm is $O(k \log n)$.

4 References

1. Ruosong Wang and David Woodruff, Tight Bounds for l_p Oblivious Subspace Embeddings, *SODA*, 2019