Adversarially Robust Streaming Algorithms (Lectures 7–9)

DS-563 / CD-543 @ Boston University
Instructor: Krzysztof Onak

Spring 2024

1 What we want to compute

Input: A stream of at most m elements from $[n] = \{1, \dots, n\}$. For simplicity, $m = O(\text{poly}(n))$.

Want to approximate: The number of distinct elements (a.k.a. F_0). Denote it $F_0(S)$ for a stream S. More specifically, for some parameter $\epsilon \in (0, 1)$, we want to output an estimate $\widehat{F}$ such that

$$(1 - \epsilon)F_0(S) \leq \widehat{F} \leq (1 + \epsilon)F_0(S).$$

Deletions: Sometimes deletions of items are allowed as well. In this setting, every stream item is of the form "insert x" or "delete x."

2 Adversarially robust streaming algorithms

Most streaming algorithms we have seen use randomness to "compress" the input into a small sketch that uses much less space than what would be needed to store the entire input. In fact, for many problems, including distinct elements, it is possible to show that an algorithm has to essentially store its entire input to be able to provide even approximate estimates if it is deterministic. Therefore, randomness is crucial for designing efficient streaming algorithms for many problems.

So far, any analysis we performed assumed that there is a fixed stream and we analyze the correctness of our estimates at the end of this stream. In some applications, however, we may want to provide continuous estimates as we go through the stream. (Think about, for instance, monitoring internet traffic.) This could be problematic if the stream is not fixed but future elements of the stream may depend on previous estimates. Estimates may leak information about the internal randomness of the algorithm. This information could be used by an adversarial actor to break our estimates. Another reason for studying this setting is that we may want to use the streaming algorithm as a subroutine for another algorithm and this type of breakage could occur accidentally.

Is there a way to make our algorithms robust to this type of information leakage? Our goal now is to discuss various techniques that allow for achieving this goal without turning to space–inefficient deterministic algorithms.

We start by introducing a model which describes the challenge we are facing.

Model: We think of our computation as a game between two players, the Algorithm and the Adversary. In each round:

- First, the Adversary sends the next element of the stream to the Algorithm.
- Second, the Algorithm sends an updated estimate to the Adversary.

Outcome: The Adversary wins if *at least one* of the estimates sent by the Algorithm is not a $(1 \pm \epsilon)$–multiplicative approximation to the current value.

Resources:

- The Algorithm should use space as small space as possible. (It's a streaming algorithm after all!)
- The Adversary can base next updates on her own random coin tosses and previous estimates of the Algorithm, She does not know the Algorithm's personal coin tosses other than through the estimates that are made public.

Our goal: turn non-robust algorithms into robust

- An *non-robust* streaming algorithm here is an algorithm about which we only know that it works with good probability for any fixed stream.
- An *(adversarially) robust* streaming algorithm is an algorithm that, with good probability, provides good estimates in the adaptive setting. That is, it wins the game described above with good probability against any adversary.

3 A simple inefficient solution: Use a different copy in each round

$\mathcal{A}$ = a non-robust algorithm that provides $(1+\epsilon)$–multiplicative approximation w.p. $1 - \delta/m$

Solution: Keep and update m independent copies of $\mathcal{A}$. In round i, return the estimate from the i-th copy.

Problem: The multiplicative overhead over $\mathcal{A}$ is $\Theta(m)$, so we could just store the entire input...

4 Technique 1: "Sketch/algorithm switching" (insertion only)

$\mathcal{A}$ = non-robust algorithm that provides $(1+\epsilon/20)$–multiplicative approximation w.p. $1 - \delta/m^2$

Algorithm 1: Insertion–only $F_0(S)$ approximation

```
1 estimate ← $0$
2 index ← $1$
3 $t = O(\epsilon^{-1}\log m)$ independent copies $\mathcal{A}_1, \ldots, \mathcal{A}_t$ of $\mathcal{A}$
4 foreach stream item $x$ do
5     pass $x$ to each $\mathcal{A}_i$ and process it independently
6     if estimate from $\mathcal{A}_{index} \geq (1+\epsilon/2)$estimate then
7         estimate ← estimate from $\mathcal{A}_{index}$
8         index ← index $+ 1$
9     output estimate
```

Observation 1 (space usage): If all estimates of $\mathcal{A}_i$'s that we look at are good (i.e., they are $(1+\epsilon)$–multiplicative approximations), then $O(\epsilon^{-1}\log m)$ copies of $\mathcal{A}$ suffice, i.e., the algorithm won't run out of the copies of $\mathcal{A}$ when increasing *index*.

Why? We switch to a new algorithm when we update our estimate. Every time we update the estimate, it increases by a factor of at least $1+\epsilon/2$. Since it can only increase from roughly 1 (after the initial 0) to at most roughly m, it cannot increase too many times.

More precisely, once we start processing the stream, the set of possible values of F_0 is $\{1,\ldots,m\}$. If all estimates are good, we only see values between $(1-\epsilon/20)$ and $(1+\epsilon/20)m$. After processing the first element, the value of variable *estimate* can increase at most $\left\lceil \log_{(1+\epsilon/2)} \frac{(1+\epsilon/20)m}{1-\epsilon/20} \right\rceil$ times. Hence the total number of copies of $\mathcal{A}$ that we need is at most

$$\begin{aligned}1+\left\lceil \log_{(1+\epsilon/2)} \frac{(1+\epsilon/20)m}{1-\epsilon/20} \right\rceil &\le 3+\left\lceil \log_{(1+\epsilon/2)} m \right\rceil \le 4+\log_{(1+\epsilon/2)} m \\ &\le 4+\frac{\log m}{\log(1+\epsilon/2)} = O\left(\frac{\log m}{\epsilon}\right).\end{aligned}$$

Observation 2 (correctness of estimates): If all estimates of $\mathcal{A}_i$'s that we look at are good, then the algorithm provides a good approximation to $F_0(\ldots)$ for all prefixes of the stream.

Why? Consider the moment in which we change the value of variable *estimate*, i.e., update our estimate. Initially, this value is a $(1+\epsilon/20)$–multiplicative approximation due to our assumption that it is good. We may, therefore, be overestimating by a factor of $1+\epsilon/20$ initially, but since the value of F_0 can only increase (as this is an insertion–only stream), we can never overestimate by more. How about underestimating? As long as we don't have to update the estimate, the current estimate from $\mathcal{A}_{\mathit{index}}$ is less than $(1+\epsilon/2)\mathit{estimate}$. Let $S_\star$ be the current prefix of the stream. Since we are assuming that the current estimate from $\mathcal{A}_{\mathit{index}}$ is good, the lowest it can be is $(1-\epsilon/20)F_0(S_\star)$. Hence we have $(1-\epsilon/20)F_0(S_\star) < (1+\epsilon/2)\mathit{estimate}$. This implies that

$$\begin{aligned}\mathit{estimate} &> \frac{1-\epsilon/20}{1+\epsilon/2}F_0(S_\star) \ge (1-\epsilon/20)(1-\epsilon/2)F_0(S_\star) \\ &\ge (1-\epsilon/20-\epsilon/2)F_0(S_\star) \ge (1-\epsilon)F_0(S_\star).\end{aligned}$$

Hence, assuming that all the estimates from $\mathcal{A}_{\mathit{index}}$ that we receive throughout the execution of the algorithm are good, we never overestimate by more than a factor of $1+\epsilon/20$ and we never underestimate by more than a factor of $1-\epsilon$. Therefore, all estimates that we output are $(1+\epsilon)$–multiplicative approximates to the evolving value of F_0.

Obstacle: So it remains to prove that these estimates are "good" for the non-robust algorithms—$\mathcal{A}_i$'s—when we obtain them. In the simple solution (Section 3), we used the fact that nothing has been revealed to the Adversary about a given $\mathcal{A}_i$ until we used it for providing an estimate and then we would immediately throw this $\mathcal{A}_i$ away and never use it again. Here, however, we make multiple queries to $\mathcal{A}_{\mathit{index}}$ to track when we cross a given threshold. This does provide some additional information to the Adversary, who knows that our estimate has not crossed the threshold and therefore, knows that some settings of internal coin tosses in $\mathcal{A}_{\mathit{index}}$ are not possible.

Getting around the obstacle:

- Assume the Adversary is deterministic. If she's not, by averaging, there must be a setting of her random coin tosses for which she manages to break our algorithm with at least the same probability.
- Prove by induction: Algorithms $\mathcal{A}_i$, for $i \in \{1, \ldots, k\}$, when we query them, give all good estimates with probability at least $1 - k\delta/m$.
  - The base case for $k = 0$ is trivially true.
  - The inductive step reasoning: Consider the moment when we set *index* to $k + 1$ and start using $\mathcal{A}_{k+1}$ to track when we cross the new threshold. Note that while we use $\mathcal{A}_{k+1}$, we keep giving to Adversary a fixed estimate, which is stored in variable *estimate*. Since Adversary is deterministic, we can simulate the stream that Adversary would produce if it received the value stored in *estimate* as our estimate till the end of the stream. $\mathcal{A}_{k+1}$ has to provide a good estimate for some prefix of this stream, until it provides an estimate that is at least $(1 + \epsilon/2)$*estimate*. Therefore, it suffices that $\mathcal{A}_{k+1}$ provides good estimates throughout this fixed stream of updates. Since this stream is fixed, it is not a problem that $\mathcal{A}_{k+1}$ is non-robust, and via the union bound, it achieves this goal with probability at least $1 - m \cdot \delta/m^2 = 1 - \delta/m$. By another application of the union bound and the inductive assumption, A_i's for $i \in \{1, \ldots, k + 1\}$ provide good approximations when queried throughout our algorithm with probability at least $1 - k\delta/m - \delta/m = 1 - (k + 1)\delta/m$.
- All internal estimates by $\mathcal{A}_i$'s and estimates sent back to the Adversary are therefore within the allowed range with probability at least $1 - m \cdot \delta/m = 1 - \delta$.

Space usage: Since a non-robust algorithm for approximating F_0 with properties as listed above needs only $O(\text{poly}(\frac{\log n}{\epsilon}))$ space, the total space is $O(\text{poly}(\frac{\log n}{\epsilon})) \cdot O(\frac{\log m}{\epsilon}) = O(\text{poly}(\frac{\log n}{\epsilon}))$.

5 Technique 2: Sparse–dense trade-offs for insertion/deletion streams

When are deletions problematic? When the number of distinct elements can change significantly very often. Sample stream: "insert 5", "delete 5", "insert 5", "delete 5", …

Technique 1 builds on the fact that the actual value cannot change significantly too often. We won't define it here formally, but this value is known as the *flip number*. Unfortunately, in the example above, the flip number is $\Omega(m)$, where m is the length of the stream.

Observation: Significant changes can only happen very often when the current number of distinct elements is small. If the number of distinct elements is $(1 \pm \epsilon/3)T$, then over the next $\epsilon T/3$ updates, T will still be a $(1 \pm \epsilon)$-multiplicative approximation to the number of distinct elements.

(Definitions) Sparsity of a vector v: We say that a vector v is *k-sparse* if at has at most k non-zero coordinates. Analogously, we say that a vector v is *k-dense* if it has at least k non-zero coordinates.

(Auxiliary Tool) Sparse recovery:

> For any k, there is a (linear sketching) streaming algorithm that uses $k \operatorname{polylog}(n)$ space and can recover all k-sparse frequency vectors over a stream of m arbitrary deletions and insertions.
>
> The algorithm provides the recovery guarantee for all vectors with probability at least $1 - 1/n^3$, where the probability is taken over the initial selection of internal coin tosses.

Note that this works for recovering k-sparse vectors even if in the meantime, the vector was arbitrarily dense.

Solution:

- Create two regimes: sparse and dense. Handle them differently.
- Sparse regime (for at most $2\sqrt{m}$-sparse vectors): Store the vector explicitly. A sparse representation uses $O(\sqrt{m})$ words. In this case, we know the number of distinct elements exactly.
- Dense regime (at least $\sqrt{m}$-dense vectors): Run in parallel (from the very beginning of the algorithm, ignoring the sparse/dense regime distinction) $3\sqrt{m}/\epsilon$ parallel copies of a non-robust algorithm that gives a $(1 \pm \epsilon/3)$-approximation with probability $1 - \delta/m$. In the dense regime, use a new unused copy every $\frac{\epsilon}{3}\sqrt{m}$ updates to give an estimate, and immediately dispose this algorithm. Stick to this estimate for $\frac{\epsilon}{3}\sqrt{m}$ updates.
- Switching between regimes:
  - From sparse to dense: When the vector becomes $2\sqrt{m}$-dense, which we track exactly, forget it and just switch to the dense regime.
  - From dense to sparse: If a new approximation of F_0 becomes lower than $\frac{3}{2}\sqrt{m}$, switch to sparse regime. We need to recover the frequency vector exactly in this case, which can be achieved using the sparse recovery tool.

Space usage: $O(\sqrt{m}) \operatorname{poly}(\epsilon^{-1} \log n)$ words

- Exact vector in the sparse regime: $O(\sqrt{m})$ words
- Sparse recovery (one instance needed): $O(\sqrt{m} \operatorname{polylog}(n))$ words
- $O(\epsilon^{-1}\sqrt{m})$ copies of a non-robust F_0 streaming algorithm: $O(\sqrt{m} \operatorname{poly}(\epsilon^{-1} \log n))$ words

Robust algorithms via differential privacy: Differential privacy is an approach to protecting data of individuals in a study of a set of people, but in this context, it can be used to protect the internal randomness of a set of copies of a non-robust algorithm. In particular, a similar space usage can be achieved by maintaining $O(m^{1/2} \operatorname{poly}(\epsilon^{-1} \log n))$ copies of a non-robust F_0 algorithm and outputting a differentially private median of their estimates. This is an essence a quadratic improve

Best currently known: $O(m^{1/3} \cdot \operatorname{poly}(\epsilon^{-1} \log n))$ words. It can be achieved by combining the sparse–dense approach presented here with the approach based on differential privacy. To achieve it, one has to lower the threshold between the sparse and dense regimes to roughly $m^{1/3}$. For the dense regime, it then suffices to maintain $O(m^{1/3} \cdot \operatorname{poly}(\epsilon^{-1} \log n))$ copies of the non-robust algorithm and to output a private median of estimates whenever an estimate is needed.

It's a great open question whether $m^{\Omega(1)}$ space is necessary for computing distinct elements on insertion/deletion streams in the adversarial setting.