CS369G: Algorithmic Techniques for Big Data Spring 2015-2016

Lecture 11: l_0-Sampling and Introduction to Graph Streaming

Prof. Moses Charikar *Scribe: Austin Benson*

1 Overview

We present and analyze the black box s-sparse recovery algorithm used in the previous lecture for l_0-sampling. The data structure for s-sparse recovery is built by combining several 1-sparse recovery data structures. Afterwards, we begin to look at graph streaming algorithms. In this framework, we are presented with a stream of edges in a graph (edges may be added or deleted) and we want to answer questions about the graph by only storing a little information per vertex.

2 Review of l_0-sampling

Given a vector $a \in \mathbb{R}^n$, an l_0-sample from a is a uniformly at random sample from the non-zero indices of a. We will be satisfied with an algorithm that samples a non-zero coordinate with probability:

$$(1 \pm \epsilon)\frac{1}{\|a\|_0} \pm \delta.$$

In the last lecture, we covered an l_0-sampling algorithm that used s-sparse recovery as a black box. Recall that, given a vector a, an s-sparse recovery algorithm will return the non-zero coordinates of a if $\|a\|_0 \leq s$; otherwise, it will return that a is not s-sparse. The algorithm works as follows. First, use a hash function $h\colon [n] \to [n^3]$,[1] and define vectors $a(j)$ by

$$[a(j)]_i = \begin{cases} a_i & \text{if } h(i) \leq n^3/2^j \\ 0 & \text{otherwise} \end{cases}$$

With this definition, $a(0) = a$, $a(1)$ has approximately $1/2$ of the coordinates zeroed out, $a(2)$ has approximately $3/4$ of the coordinates zeroed out, and so on.

In the algorithm, each vector $a(j)$ serves as an input to an s-sparse recovery routine for $s = O(\log 1/\delta)$. We choose a coordinate uniformly at random from the first vector for which the recovery succeeds, i.e., the smallest j for which s-sparse recovery succeeds on $a(j)$. Last time, we showed that there exists a j for which $s/4 \leq \|a(j)\|_0 \leq s/2$ with high probability.

We did not show last time how to select a non-zero coordinate uniformly at random. If our hash function h were truly random, then we could simply take the index i returned by the recovery procedure that minimizes $h(i)$. While we don't have a truly random hash, selecting from a k-wise independent family of hash functions is sufficient, as the following theorem says.

[1] We hash to $[n^3]$ because with high probability, the hash will be collision-free.

Theorem 1. *Let $k = O(\log(1/\epsilon))$ and suppose that h is drawn uniformly at random from a family of k-wise independent hash functions. Then for any non-zero coordinate j of a,*

$$\Pr\left(\min_{i:a_i \neq 0} h(i) = j\right) = (1 \pm \epsilon)/\|a\|_0.$$

3 Sparse recovery

The major question is how do we implement the s-sparse recovery routine? While we used count-min for s-sparse recovery before, the algorithm was not able to recover the actual non-zero indices of the vector. We will need this for l_0-sampling.

3.1 1-sparse recovery: first attempt

We begin with a simple algorithm for 1-sparse recovery. In 1-sparse recovery, we are given a vector a in a streaming fashion (where updates may be positive or negative) and we want to determine if the vector has exactly one non-zero entry. Specifically, at any point in the stream, the algorithm should be able to say:

1. the vector a has a single non-zero coordinate i with value a_i (recovery); or,
2. the vector a does not have a single non-zero coordinate (failure).

For now, we will make two assumptions on the data and the stream. First, the coordinates and updates are all integers. Second, there are only positive updates. With the second constraint, there are simple algorithms for 1-sparse recovery because only one index may appear in the stream; however, we will relax this constraint later and use the ideas from this setting.

Let a_i represent the updates in the stream. Consider the following algorithm. Keep track of

- $w_1 = \sum a_i$,
- $w_2 = \sum i a_i$, and
- $w_3 = \sum i^2 a_i$.

We then estimate the non-zero index by w_2/w_1 with value w_1. It is straightforward to see that if a is indeed 1-sparse, then this ratio provides the correct index as there are no negative updates. To check whether or not a is 1-sparse, we simply check for consistency with w_3, i.e., test if $w_3 = (w_2/w_1)^2 w_1$.

Next, we allow for negative updates in the stream. Again, if a is 1-sparse the previous algorithm will work. However, due to cancellation, we cannot determine when a is not 1-sparse.

3.2 1-sparse recovery: second attempt

We consider an alternative method for the consistency check. Suppose now that our algorithm keeps track of w_1 and w_2 as before as well as

$$w_3 = \sum a_i z^i \mod p$$

for some large prime $p > N$ and some z chosen uniformly at random from $\{0, \ldots, p-1\}$. We then test if

$$w_3 \equiv w_1 \cdot z^{w_2/w_1} \mod p \tag{1}$$

Note that if $w_1 = 0$, then either all updates are 0 or there was cancellation. In either case, a is not 1-sparse. So we can just consider the case when $w_1 \neq 0$. Also, it is easy to see that if a is 1-sparse, the test in Equation 1 will be true.

Lemma 2. *If a is not 1-sparse, then the test in Equation 1 fails with probability less than n/p.*

Proof. In order for the test to succeed, we need that

$$\sum a_i z^i - w_1 z^{w_2/w_1} \equiv 0 \mod p$$

Note that the left-hand-side is a degree n polynomial. Thus, if a is not 1-sparse, the polynomial can only be 0 if z is a root. A degree n polynomial can only have n distinct roots, and z was chosen uniformly at random from p numbers. Thus, the probability of z being a root (and the test failing) is at most n/p. □

We now have an algorithm for 1-sparse recovery:

1. if the vector a is 1-sparse, the algorithm returns the index (w_2/w_1) and value (w_1), and
2. if the vector a is not 1-sparse, then the algorithm returns failure with high probability (at least $1 - n/p$).

3.3 s-sparse recovery

We use 1-sparse recovery as a black box to build our s-sparse recovery data structure. Let there be $2s$ buckets so that we hash the coordinates of a into one of the $2s$ buckets uniformly at random. For each bucket, we run 1-sparse recovery on the coordinates hashed to that bucket. If a is truly s-sparse, then the probability that any given non-zero coordinate lands in a bucket by itself is at least $1/2$.

Now we make $\log(s/\delta)$ copies of the above data structure. Then the probability that a non-zero coordinate is not recovered by any of the $\log(s/\delta)$ copies is at most $(1/2)^{\log(s/\delta)} = \delta/s$.

Putting everything together, our s-sparse recovery algorithm aggregates the non-zero coordinates from the $2s \cdot \log(s/\delta)$ buckets. If it finds s or fewer coordinates, then a is indeed s-sparse.

Finally, it could be the case that a has more than s entries and we only recover a subset of s or fewer. We can avoid this by keeping an additional fingerprint of a with little cost [1].

4 Graph streaming

In graph streaming, the stream consists of a sequence of edges. In general, these edges can be additions (appearing) or subtractions (disappearing), but for now, we will focus on the case where edges are only appearing.

4.1 Connected components

Our first graph streaming algorithm is connected components. For this, we simply keep buckets of vertices for each connected component. When a new edge (u, v) arrives, one of three things can happen:

1. If we haven't seen u or v, then we start a new bucket with nodes u and v.
2. If we have seen u but not v, then add v to u's bucket.
3. If u and v are in different buckets, then merge those buckets.

The connected components are the buckets, and we have only stored $O(n)$ data.

4.2 Shortest paths

Next, we look at shortest paths, and the algorithm is also straightforward to describe. We use the graph H produced by the following streaming algorithm to compute shortest paths.

Algorithm 1 Graph streaming shortest paths sketch

```
H ← ∅
for (u, v) in stream do
    if d_H(u, v) > α then        // d_H is shortest path distance in H
        H ← H ∪ (u, v)
    end if
end for
return H
```

While the algorithm is simple, we still have a couple questions to answer:

1. How close are the shortest path distances in H to the true shortest path distances?
2. How much space does the algorithm use?

The first question is straightforward.

Lemma 3. *Let G be the graph consisting of all edges in the stream. Then*

$$d_G(s, t) \leq d_H(s, t) \leq \alpha d_G(s, t)$$

for any nodes s and t.

Proof. The first inequality is clear since $H \subset G$. Any edge (u, v) in a shortest path between s and t in G would be included in H unless there is a path from u to v of distance at most α. Thus, the length increases by a factor of at most α for each edge. □

A graph H satisfying the hypothesis of the above lemma is called an *α-spanner* of G.

The *girth* of a graph is the length of the shortest cycle or ∞ if the graph has no cycles. To answer the space question, we first note that H has large girth.

Lemma 4. *The graph H has girth at least $\alpha + 2$.*

Proof. Suppose that H had a cycle of length at most $\alpha + 1$. Consider the last edge (u, v) added to the cycle from the stream. Then $d_H(u, v) \leq \alpha$ because there are at most α edges that constitute the rest of the cycle. Hence, (u, v) would not be added to H, and we have a contradiction. □

The following theorem says that large-girth graphs cannot have too many edges.

Theorem 5. *A graph with girth $g = 2t+1$ has $O(n^{1+1/t})$ edges, where n is the number of vertices.*

Proof. Let m be the number of edges in the graph and let $d = 2m/n$ be the average degree. Consider the $(d/2)$-core of the graph which results from iteratively removing all nodes with degree less than $d/2$. We claim that some vertex is left after this procedure. Each removal throws out strictly fewer than m/n edges. Thus, if we removed all vertices, we would have thrown out fewer than $(m/n) \cdot n = m$ edges, which is a contradiction.

Consider any one of the remaining nodes. We claim that we can construct a tree of depth t by performing t steps of breadth-first search. The result of the BFS must be a tree because the graph has girth $2t + 1$, so any loops would cause a cycle of size smaller than the girth. We are able to get a tree of depth t because every node has degree greater than 1 (it survived the pruning). Thus, the BFS never stops at any node in this procedure.

This tree has at least $(d/2 - 1)^t$ nodes, so

$$(d/2 - 1)^t \leq n \rightarrow (m/n - 1)^t \leq n \rightarrow m \leq n^{1+1/t} + n.$$

□

Plugging in $\alpha = 2t - 1$, we see that our space requirement is only $O(n^{1+2/(\alpha+1)})$.

References

[1] Cormode, Graham, and Donatella Firmani. A unifying framework for ℓ_0-sampling algorithms." *Distributed and Parallel Databases* 32.3 (2014): 315-335.