COMP 480/580 — Probabilistic Algorithms and Data Structure Jan 31, 2018

Lecture 8: Stream Computing and Reservoir Sampling I

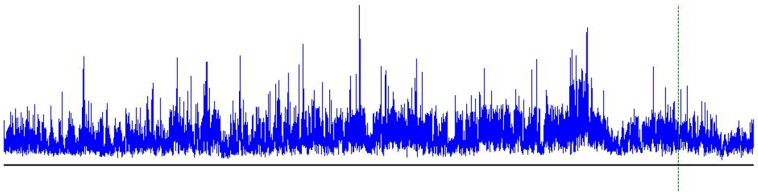
Lecturer: Anshumali Shrivastava *Scribe By: Seth Davis*

1 Data Streams: Basics

We will want to consider computation on **data streams**, (generally large) sets of data that we **do not know in advance.** Good examples of data streams are Google search queries or trending items on Twitter. These huge data sets are worthwhile to study - tracking Google queries for flu symptoms allows for efficient tracking of the flu virus, for example.

In applications involving data streams, the data often comes at an overwhelmingly fast rate and there's know way to know the size of the data in advance. For these reasons, it is convenient to think of the stream as a data set of infinite size. We'll make this notion more formal below, but the punchline is that we **cannot store the data**. (We usually only have 10-20 GB to make a calculation or otherwise draw inference.)

The key question is then: "How do you make critical calculations about the stream using a limited amount of memory?"



2 Applications

We list several examples of important applications:

1. **Mining query streams:** Google wants to know what queries are more frequent today than yesterday.
2. **Mining click streams:** Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour.
3. **Mining social network news feeds:** E.g., look for trending topics on Twitter, Facebook.
4. **Sensor networks:** Many sensors feeding into a central controller.
5. **Telephone call records:** Data feeds into customer bills as well as settlements between telephone companies.
6. **IP packets monitored at a switch:** Gather information for optimal routing and detect denial-of-service attacks.

3 Formalization: One pass model

Here we formalize the above-motivated ideas with the "one pass model":

At time t, we observe the new data point x_t, so that we have observed the sequence $D_t = \{x_1, x_2 ... x_{t-1}, x_t\}$. We do not know in advance how many data points we will observe. At all points in time we have a limited memory budget, which is far less than required to store D_t in its entirety. Our goal is to calculation $f(D_t)$, for some function f. At any point in time t, the algorithm must be able to compute (or approximate) $f(D_t)$ given x_t and the previous state from time $(t-1)$, since we cannot store data.

We will now use the above formalism to try and address some basic, practical questions about drawing inference from data streams.

4 Basic Question: Sampling

If we can get a representative sample of a data stream, then we can do analysis on it and estimate statistics of the whole stream. So, we start with the fundamental question of how to sample a stream.

4.1 Example: Finite size sampling

Say that we have seen $\{x_1, ...x_{1000}\}$. (In this case, we *do* know the size of the data in advance.) Our memeory can only store 100 of the examples. We thus want to try and sample 10% of the data stream.

1. Attempt 1: Store every tenth data entry. This is a good starting point, but it assumes that the data is uniformly distributed.

2. Attempt 2: Every time we store a sample, we generate a random number in [1-10] that determines how many points we skip until we reach our next sample.

This example ignores some of the most challenging issues of data stream computation. If the size of the stream is unbounded, the above algorithms will run unbounded. The above algorithms were also designed with the explicit ratio of storage size to data size in mind, which we often do not have. Finally, one should also think about whether a sampling algorithm is introducing sampling bias.

4.2 Example: Sampling duplicates vs originals

Say we have a data set with U unique elements and $2D$ elements that come in duplicate pairs. (For a total of $N = U + 2D$ elements.) We are interested in estimating the fraction of the elements that have duplicates. Of course, we know that the correct answer is

$$\text{Fraction of duplicates} = \frac{2D}{U + 2D}. \tag{1}$$

However, say that we take a random sample of 10% of the data and try to estimate this quantity. On expectation, the sample will contain $U/10$ of the unique elements and $(2D)/10$ of the duplicated items. However, if the sample is truly random, there is only a $1/100$ chance that an item *and* it's duplicate are both in the sample. Thus, the sample will dramatically underestimate the fraction of the items that are duplicates. How many duplicates will it under

count? We expect $D/100$ pairs of duplicates in our sample, so $(2D)/100$ of the $(20D)/100$ duplicates will be correctly identified. This leaves $(9D)/10$ duplicates that are mistakenly assumed to be unique elements. We would then estimate that

$$\text{Fraction of duplicates} = \frac{2D}{10U + 20D}. \tag{2}$$

Our estimate is only 1/10 the correct answer.

5 Reservoir Sampling

5.1 Goals

We want to sample s elements form a stream. At the time we stop at the n^{th} element, we want the following:

1. Every element has exactly s/n probability of being in the sample.
2. We have exactly s elements sampled.

We want to know if this can be done. It turns out that it can, with the technique known as reservoir sampling. Before we go straight into reservoir sampling, we can try to guess how the above would have to be accomplished, assume it's even possible.

5.2 Intuition

By assumption, at time $n-1$, we have s elements sampled, and every element previously seen has a $s/(n-1)$ probability of being in the new sample. At time n, we see the new element x_n. We want to have a s/n chance of including x_n into the sample. We also have to enforce a s/n chance of being included on all the previously seen elements, many of which we note are long gone! Since $s/n < s/(n-1)$, we have to include a way of randomly throwing out previously-stored samples.

The algorithm must then have the following form: At time n we see the new element x_n. We randomly decide if x_n is sampled. If x_n is sampled, then we must randomly select a previously sampled element to discard.

If an algorithm that accomplishes our goals exists, it must have the above form. But can the random decisions in the above protocol be made to conform to the desired probabilities? It turns out that they can, as we now describe.

5.3 Protocol: Reservoir sampling of size s

We sample elements with the following protocol:

1. Observe x_n
2. if $n < s$: keep x_n
3. else: with probability s/n, select x_n. Then choose (uniformly) one of the previously sampled elements and replace it with x_n

We claim that at any time n, any element in the sequence $\{x_1, x_2, ..., x_n\}$, has exactly an s/n chance of being in the sample.

5.4 Proof: Induction

We prove the above claim by induction. Our **induction hypothesis** is that after observing n elements, the sample S contains each element seen so far with probability s/n.

Our induction hypothesis is thus that the algorithm has produced the desired result to step n.

At time $n+1$, we see a new data point, x_{n+1}. The **induction step** of the proof will be to prove that if we implement the algorithm at time $n+1$, we will get the desired result at step n+1.

We implement the algorithm. By construction, x_{n+1} is included with probability $s/(n+1)$.

For an element x_j already in S, the probability that the algorith leaves it in S for the next round is

$$P[x_j\text{survives}] = P[x_{n+1} \text{ rejected}] + P[x_{n+1} \text{ accepted}] * P[x_j \text{ not selected for deletion}] \quad (3)$$

$$= \left(1 - \frac{s}{n+1}\right) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right) = \frac{n}{n+1} \quad (4)$$

Then, the probability that x_j is in the sample at time $n+1$ is

$$P[x_j \text{ in } S \text{ at } n+1] = P[x_j \text{ in } S \text{ at } n] * P[x_j\text{survives}] \quad (5)$$

$$= \frac{s}{n}\frac{n}{n+1} = \frac{s}{n+1}, \quad (6)$$

as desired, completing the proof.

6 Weighted Reservoir Sampling

6.1 Problem definition

We now consider a generalized version of the previous problem, **weighted reservoir sampling**. In this case, every element x_i has a corresponding weight, w_i, which is a positive real number. As before, we would like to sample elements form the stream such that at time n, we always have s elements sampled. However, we would now like to base the sample probability off of the elements weights, so that each element x_i in the sample has the sample probability

$$P\left[x_i \in S\right] = \frac{s\ w_i}{\sum_{j=1}^{n} w_j}. \quad (7)$$

We note that if all elements have weight $w_i = 1$, we recover the basic reservoir sampling situation.

6.2 Solution

The solution here is less trivial. It was given in 2006 by Pavlos S Efraimidis and Paul G Spirakis. They recommend sampling with the following protocol:

1. Observe x_n
2. Generate r_n uniformly in [0-1]
3. Set score $\sigma_i = r_i^{1/w_i}$.
4. Keep the sample as the elements with the s highest scores.

6.3 Some important points

We postpone the discussion of why of the weighted reservoir sampling algorithm works to the next section and discuss the above protocol.

The first thing to point out is that to be able to effectively maintain a list of the top s scores, should maintain a heap of element-score tuples, (x_i, σ_i) so that we can find and delete the lowest score in $\mathcal{O}(\log(s))$ time. We note that this is slightly slower than our previous sampling routine, which enjoyed $\mathcal{O}(1)$ insertion and deletion. (But we aren't worried, because it's logarithmic and s doesn't have to be huge.)

We also note that there is an important floating point arithmetic problem that one should be careful about. We note that by construction, the scores are on the interval $\sigma_i \in (0, 1)$. As we run our algorithm for a long time, we would expect to accumulate top scores that get closer and closer to all being 1. As time goes on, this would eventually lead to a floating point issue, since we are trying to deal with s numbers all so close to 1 that they are hard to distinguish. While we discussed this issue in class, we did not give an explicit solution, but one can imagine using tricks like logarithmic mappings to try and overcome it.

We can also ask how one should perform estimation on a stream. For example, one could ask "How many unique elements have I seen so far?". This problem could be solved effectively by keeping a Bloom filter - the Bloom filter has minimal memory usage and doesn't get bogged down counting duplicates. We could at any time use the number of flipped bits in the Bloom filter to easily estimate the number of unique items that we have seen.

6.4 Why it works

We close the lecture with a discussion of why the weighted reservoir sampling algorithm works. We will not give a full proof, but will prove the following helpful lemma which makes the statement plausible:

"Given random variables $\{r_1, ...r_n\}$, uniformly distributed on the interval $[0, 1]$, and weights $\{w_1, ..., w_n\}$, then

$$P\left[r_n^{1/w_n} \geq \max_{j<n}\left[r_j^{1/w_j}\right]\right] = \frac{w_n}{w_1 + ... + w_n}." \tag{8}$$

To see this, we note that, (for each j), since r_j is uniform, it's probability density function is simply 1 on the interval $[0, 1]$. We want the probability that $r_j < r_n^{w_j/w_n}$ for each j, and we can see that this is the same as the expectation value of the indicator function

$$\hat{I}(r_1, ...r_n) = \prod_{j<n} \Theta(r_n^{w_j/w_n} - r_j), \tag{9}$$

where

$$\Theta(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases} \tag{10}$$

is the *Heaviside* function. To motivate this, we note that we want to integrate over all possible values of the r variables, but only count the ones where r_n has the highest score, which only

happens when all the Θ's are 1. We then have

$$P\left[r_n^{1/w_n} \geq \max_{j<n}\left[r_j^{1/w_j}\right]\right] = \int_0^1 dr_n \prod_{j<n}\left(\int_0^1 dr_j\ \Theta(r_n^{w_j/w_n} - r_j)\right) \tag{11}$$

$$= \int_0^1 dr_n \prod_{j<n}\left(\int_0^{r_n^{w_j/w_n}} dr_j\ 1\right) \tag{12}$$

$$= \int_0^1 dr_n \prod_{j<n}\left(r_n^{w_j/w_n}\right) \tag{13}$$

$$= \int_0^1 dr_n\ r_n^{(1/w_n)\sum_{j<n} w_j} \tag{14}$$

$$= \frac{1}{1 + (1/w_n)\sum_{j<n} w_j} \tag{15}$$

$$= \frac{w_n}{w_1 + ... + w_n} \tag{16}$$

$$\tag{17}$$

References