Dealing with Massive Data January 31, 2011

Lecture 2: Distinct Element Counting

Lecturer: Sergei Vassilvitskii *Scribe:Ido Rosen & Yoonji Shin*

1 Introduction

We begin by defining the stream formally.

Definition 1 *A finite data stream, $\mathcal{X} = x_1x_2x_3...x_n$, is a sequence of n items chosen from a set of $m \leq n$ unique elements, where elements may be repeated, without assuming a specific underlying distribution. An (offline) adversary may be generating this stream, such that the adversary may see your algorithm, think for a while, and generate the worst possible stream if he/she so desires. (We assume the adversary is oblivious, that is he generates the stream without seeing the outcome of the probabilistic decisions made by the algorithm.)*

Some properties of algorithms running on data streams are:

- Sees inputs only once, in sequential order.
- No random access over inputs, except those stored in (relatively small) memory.
- Small, limited amount of memory; can not store everything. (Aim for $O(\log n)$ or smaller.)

Data streams are useful when:

- There is too much much data (information), and thus not enough time or space to cope.
- Sequential access of data (i.e. streaming) is orders of magnitude faster than accessing non-sequential data in most cases since no seeking is involved (such as when it is stored on disks or disk arrays), and since reads are predictable and easily cached when reading sequentially. (i.e.: I/O bounded algorithms.)
- Sometimes it is not necessary to save all the data, e.g. when counting the number of events.

2 How many distinct elements are in a stream?

Let's start with a simple task:

Problem 1 *Given a stream $\mathcal{X} = x_1, x_2, x_3, ..., x_n$ of distinct elements $a_1, a_2, ..., a_k$. Let $D(\mathcal{X})$ be the number of distinct elements. Find $D(\mathcal{X})$ under the constraints for algorithms on data streams.*

Idea 1 *Build a **hash table**.*

This requires $O(n)$ space in the worst case, which we do not have.

Remark 1 *Whether or not we know how big the stream is ahead of time does not matter.*

Idea 2 *Maintain a* ***sample****.*

We will show that we cannot sample uniformly across the stream and expect to get accurate results.

Example 1 *Consider the two possible streams, with $n \gg k$:*

$$\mathcal{X}_1 = \underbrace{a_1 a_1 a_1 ... a_1}_{n \text{ times}} ...$$
$$\mathcal{X}_2 = \underbrace{a_1 ... a_1}_{n-k} \underbrace{a_2 a_3 ... a_{k+1}}_{k} ...$$

These are hard to differentiate by maintaining a uniform sample across the data.

In the first scenario we have $D(\mathcal{X}_1) = 1$, whereas in the second we have $D(\mathcal{X}_2) = k + 1$, but a random sample from the two streams may look identical. In the case that the sample looks the same, we will not be able to distinguish between the two streams and must err when reporting the error on one of them. Suppose we sampled $r < n$ elements from $\mathcal{X}_2$. What is the probability that the sample only has elements a_1?

Let $s_1, s_2, \ldots, s_r$ be the sampled elements. We have:

$$\begin{aligned}
\Pr[s_1 = a_1] &= \frac{n-k}{n} \\
\Pr[s_2 = a_1 | s_1 = a_1] &= \frac{n-k-1}{n-1} \\
&\vdots \\
\Pr[s_i = a_1 | s_1 = a_1, ..., s_{i-1} = a_1] &= \frac{n-k-(i-1)}{n-(i-1)} \\
\therefore \Pr[\text{bad event}] &= \Pr[s_1 = a_1] \cdot \Pr[s_2 = a_1 | s_1 = a_1] \cdots \Pr[s_r = a_1 | s_1 = a_1, ..., s_{r-1} = a_1],
\end{aligned}$$

where the last conclusion follows because the samples are drawn independently.

$$\begin{aligned}
\therefore \Pr[\text{bad event}] &= \prod_{i=1}^{r} \left(\frac{n-k-(i-1)}{n-(i-1)} \right) \\
&\geq \prod_{i=1}^{r} \left(\frac{n-k-r}{b-r} \right) \geq \left(\frac{n-k-r}{n-r} \right)^r = \left(1 - \frac{k}{n-r} \right)^r \\
&\geq e^{-\frac{2kr}{n-r}} \qquad \text{(by the useful inequality, below)}
\end{aligned}$$

Fact 1 *For any small $0 \leq z \leq \frac{1}{2}, (1-z) \geq e^{-2z}$.*

Now we have a useful lower bound on the probability of a worst-case sample (which would result in a worst-case answer of $D(\mathcal{X}) = 1$). This lower bound is very bad even for large samples.

Choosing generous parameters $r = \frac{n}{10}$, $k = 4$ (5 distinct elements in the stream), we have:

$$\Pr[\text{bad event}] \geq e^{-\frac{2kr}{n-r}} \geq e^{-\frac{8/10^n}{9/10^n}} \geq e^{-\frac{8}{9}} (\approx 41\%)$$

This is very bad, especially given that we are sampling 1/10th of the input!, we will err by at least a factor of 2 almost half the time.

Let's try another route...

3 The magic of hashing

First, consider a simplification of 1. We are given some number t, and are asked, is $D(\mathcal{X}) \gg t$ or $D(\mathcal{X}) \ll t$?

Problem 2 *Given a threshold, t, and a stream, $\mathcal{X} = x_1, x_2, ..., x_n$,*

$$\textit{If } D(\mathcal{X}) \left\{ \begin{array}{l} \geq t \\ < \frac{t}{2} \\ \in [\frac{t}{2}, t] \end{array} \right\}, \textit{ then output } \left\{ \begin{array}{c} \textit{yes} \\ \textit{no} \\ \textit{anything} \end{array} \right\}, \textit{ with probability } \geq 90\%.$$

Suppose we have an **ideal hash function** $h : X \mapsto [1, t]$. By definition, $\Pr[h(x) = i] = 1/t$.

Algorithm 1 Noisy counter

```
for all x_i ∈ X do
    if h(x_i) = t then
        return YES.
return NO.
```

Algorithm 1 doesn't quite work to reach our 90% probability goal, but can be boosted (later).

Proof. Suppose $D(\mathcal{X}) \geq t$, if we consider the probability of failure, that we will return "NO" if the answer was supposed to be "YES". Recall that $\Pr[h(a_i) = t] = 1/t$, therefore $\Pr[h(a_i) \neq t] = 1 - \frac{1}{t}$. We repeat this experiment $D(\mathcal{X})$ times to obtain:

$$\begin{aligned} \Pr[\text{failure}] &= \underbrace{\left(1 - \frac{1}{t}\right)\left(1 - \frac{1}{t}\right)\cdots\left(1 - \frac{1}{t}\right)}_{D(\mathcal{X}) \text{ times}} \\ &= \left(1 - \frac{1}{t}\right)^{D(\mathcal{X})} \leq \left(1 - \frac{1}{t}\right)^{t} < \frac{1}{e} (\approx 37\%) \\ \therefore \Pr[\text{success}] &> 60\% \end{aligned}$$

Therefore, when $D(\mathcal{X}) \geq t$, Algorithm 1 is correct with more than 60% probability.

Now suppose $D(\mathcal{X}) < t/2$, and consider the probability of success, that we will return "NO" if the answer was indeed "NO". We have that $\Pr[h(a_i) \neq t] = 1 - \frac{1}{t}$, so in this case for success we have:

$$\Pr[\text{success}] = \underbrace{\left(1-\frac{1}{t}\right)\left(1-\frac{1}{t}\right)\cdots\left(1-\frac{1}{t}\right)}_{D(\mathcal{X}) \text{ times}}$$

$$= \left(1-\frac{1}{t}\right)^{D(\mathcal{X})} \geq \left(1-\frac{1}{t}\right)^{t/2} > 60\% \qquad \text{(approximately)}$$

When $D(\mathcal{X}) < t/2$, Algorithm 1 is correct with more than 60% probability.

Thus, for both cases that matter, Algorithm 1 produces results with greater than 60% accuracy. □

How can we further amplify the signal from Algorithm 1 to reach our goal of 90%?

4 Boosting the success probability

Idea 3 *Run Algorithm 1 k times (in parallel) using different,* ***independent*** *hash functions, $h_{j\in\{1...k\}}$. If more than 1/2 of the functions say YES, then output YES. Otherwise, output NO.*

Remark 2 *$h_1, ..., h_k$ must be independent of eachother so that we can use the Chernoff Bound below.*

Proof. Let Z_i be the event that the ith counter (Algorithm 1 using h_i) said YES. Then, $\Pr[Z_i] \geq 0.6$ from before. Let $Z = \sum_{i=1}^{k} Z_i$. The expected value of Z is $E[Z] = 0.6k$ by definition. We care about the probability that a majority of counters will fail simultaneously: $\Pr[Z \leq \frac{k}{2}]$.

$$\Pr[Z \leq k/2] = \Pr\left[Z \leq \frac{0.5}{0.6}E[Z]\right]$$

Recall from the previous lecture the Chernoff Bound, which in this case is formulated as:

$$\Pr[Z \leq (1-\delta)E[Z]] \leq e^{-\frac{E[Z]\delta}{2}} \qquad \text{(Chernoff Bound)}$$

Using the Chernoff bound above we have, finally:

$$\Pr[Z \leq k/2] \leq e^{\left(-\frac{1}{2}\cdot 0.6k\cdot\left(\frac{0.1}{0.6}\right)^2\right)}$$

Therefore, for a success probability of $1-q$, we need to use $k = O(\log\frac{1}{q})$ independent hash functions, and thus we need $O(\log\frac{1}{q})$ independent counters in the boosted counter that Idea 3 constructs. □

We have solved problem 2, of thresholding the number of distinct elements in a stream, with high (as we desire) probability. How can we then count distinct elements in a stream, as in problem 1?

5 Bringing it all together

First, we produced a single counter, as in Algorithm 1, given an ideal hash function h, to determine if a stream has $> t$ or $\leq t/2$ distinct elements with some confidence ($\geq 60\%$). Then, we boosted this counter using multiple hash functions to obtain the desired probability $(1-\delta)$ as a function of k, the number of independent hash functions $(h_1, ..., h_k)$ we used in parallel. This boosted counter is described in idea 3. Now, we can now use multiple, independent boosted counters to peg the number of distinct elements in the stream $\mathcal{X}$, $D(\mathcal{X})$, into as small a range as memory permits...

Idea 4 *Using the boosted counters from idea 3, we now set multiple thresholds $t = 2, 4, 8, 16, ..., n$, and use $|\{t\}|$ many counters. Using powers of 2 up to n yields about $O(\log n)$ thresholds. Each counter needs only one bit of memory to store its result (YES/NO), so we only use $O(\log n)$ space.*

6 Uniform, random sampling of distinct elements in a stream

In addition to simply knowing the number of distinct elements, it may be useful to maintain a uniform sample of the distinct elements.

Problem 3 *Given a stream $\mathcal{X} = x_1 x_2 ... x_n$, now let $D(\mathcal{X})$ be the set of distinct elements in the stream $\mathcal{X}$. (For example, for the stream $\mathcal{Y} = a_1 a_1 a_1 a_2 a_1 a_3 a_1 a_1 ...$, $D(\mathcal{Y}) = a_1, a_2, ..., a_{k+1}$.) Return a random, uniform sample from $D(\mathcal{X})$ of some fixed size M.*

Remark 3 *Due to time constraints in lecture, we sketch a solution, leaving analysis for next time.*

Suppose we have a hash function, $g : X \mapsto [0, m-1]$, that maps elements of $\mathcal{X}$ onto the range from 0 to $m-1$. Suppose we further have a function, $h(x)$, which returns the leading 0s in the base two representation of $g(x)$. For example, say $n = 16$, so h is represented in 4 bits. Take $g(a_1) = 4$ which is 0100 in base two, thus $h(a_1) = 1$. For $g(a_2) = 9$ (1001 in base 2), $h(a_2) = 0$. It is easy to see that:

$$\begin{aligned} \Pr[h(x) = 0] &= \frac{1}{2} \\ \Pr[h(x) = 1] &= \frac{1}{4} \\ \Pr[h(x) = 2] &= \frac{1}{8} \\ &\vdots \\ \Pr[h(x) = i] &= \frac{1}{2^{i+1}} \end{aligned}$$

How can we use this biased hash function, h, to achieve our goal?

Idea 5 *Store elements in different "levels" based on value of h for that element, and increase the minimum required level as the number of elements stored exceeds memory.*

Algorithm 2 Online sampling of distinct elements in a stream

Let M be the maximum amount of memory we may use.
Let h be the weird hash function with the properties described above.

current level $\leftarrow 0$
for all $x \in X$ **do**
  level $\leftarrow h(x)$
  if level $\geq$ current level **then**
    add x to sample S at level $h(x)$
  if $|S| \geq M$ (i.e. out of memory) **then**
    current level $\leftarrow$ current level $+1$
    only keep elements in S whose $h(x) >$ current level

Observe that for Algorithm 2:

- The number of distinct elements at level i is $D(\mathcal{X})/2^{i+1}$.
- The number of distinct elements at level i or higher is $D(\mathcal{X})/2^{i}$.
- At some level, there is a perfect tradeoff in terms of number of elements in the sample.