6.897: CSCI8980 Algorithmic Techniques for Big Data September 12, 2013

Lecture 2

Dr. Barna Saha *Scribe: Haritha Bellam*

Overview

Till now we have looked at streaming algorithms which approximately count the number of distinct items in a stream. In this lecture, we shall look at another fundamental problem of finding approximate frequency of any item in the stream. A celebrated technique (known as count-min sketch) from the streaming world shall be presented to answer the approximate frequency and related queries.

1 Introduction

Consider an n-dimensional vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]$. Initially all the entries in $\mathbf{a}$ are zero, i.e., $a_i = 0$. Let X be the stream of m updates, where the tth update is represented as (i_t, c_t). The tth update implies that $a_i \longleftarrow a_i + c_t$ and the rest of the entries are unchanged.

After any number of updates (say t), some of the queries of interest are the following:

1. *Point Query:* Given i; return an approximation of a_i.
2. *Range Query:* Given l, r; return an approximation of $\sum_{i=l}^{r} a_i$.
3. *Heavy Hitters Query:* Given $\phi \in (0, 1)$; a heavy hitter is an item $a_i \geq \phi \|a\|_1$. The goal is to return all the approximate heavy hitters i s.t. $a_i \geq (\phi - \epsilon)\|a\|_1$.
4. *Quantile Query:* Given $\phi \in (0, 1)$; return a j s.t. $(\phi - \epsilon)\|\mathbf{a}\|_1 \leq \sum_{i=1}^{j} a_i \leq (\phi + \epsilon)\|\mathbf{a}\|_1$.

2 Count-Min Sketch

Data Structure: Initialize a two-dimensional array *count* with width w and depth d. Each entry in the array is initially zero. Given parameters (ϵ, δ), set $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil \ln \frac{1}{\delta} \rceil$. Also, d hash functions $h_1, \ldots, h_d : \{1, 2, \ldots n\} \rightarrow \{1, 2, \ldots, w\}$, are chosen uniformly at random from a pairwise-independent family.

Handling an Update: When an update (i_t, c_t) arrives, then c_t is added to one entry in each row of the array *count*. Specifically, $\forall 1 \leq j \leq d, count[j, h_j(i_t)] \leftarrow count[j, h_j(i_t)] + c_t$.

Lemma 1. *The space used by Count-Min Sketch is $O(wd) \equiv O(\frac{1}{\epsilon} \ln \frac{1}{\delta})$ words. Specifically, it uses an array which takes wd words and d hash functions, each of which can be stored using 2 words. An update can be handled in $O(d) \equiv O(\ln \frac{1}{\delta})$ time.*

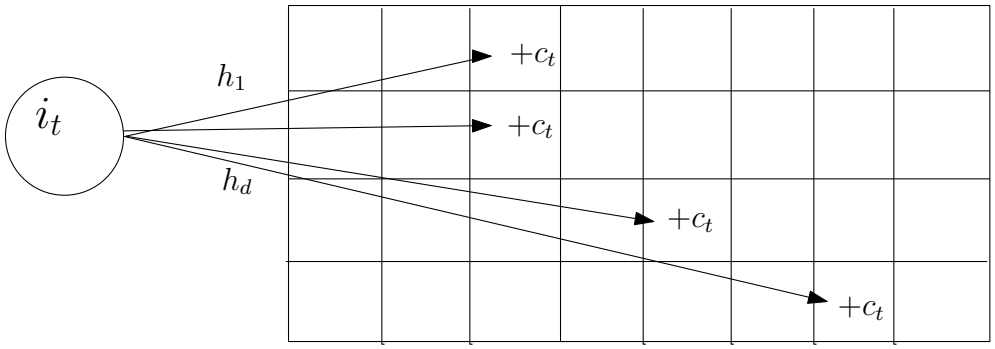

Figure 1: An item i mapped to one cell in each row.

3 Answering Queries

3.1 Point Query

Given i, the answer returned is $\hat{a_i} = \min_{j=1}^{d} count[j, h_j(i)]$.

Lemma 2. *The time answer the point query is $O(\ln \frac{1}{\delta})$. The estimate $\hat{a_i}$ has the following two properties:*

1. $a_i \leq \hat{a_i}$.
2. $\hat{a_i} \leq a_i + \epsilon \|a\|_1$ *with probability at least* $1 - \delta$.

Proof. Consider the jth hash function h_j. Consider the array entry $count[j, h_j(i)]$ into which any update to item a_i happens. Since we assume that at each update $c_t \geq 0$, it follows that $a_i \leq \hat{a_i}$. Now we want to find the "excess quantity" which has been stored at $count[j, h_j(i)]$.

Define an indicator variables $I_{i,j,k}$ which are 1 if $(i \neq k) \wedge (h_j(i) = h_j(k))$, and 0 otherwise. By pairwise independence of the hash functions, then

$$E(I_{i,j,k}) = Pr[h_j(i) = h_j(k)] \leq 1/\text{range}(h_j) = \frac{\epsilon}{e}$$

where range(h_j) is the range of values the hash function h_j can take. Now define the variable $X_{i,j}$ (random over the choices of h_i) to be $X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} a_k$. ($X_{i,j}$ is the excess quantity which has been stored at $count[j, h_j(i)]$.) By construction of the data structure, $count[j, h_j(i)] = a_i + X_{i,j}$. Therefore,

$$
\begin{aligned}
E(X_{i,j}) &= E(\sum_{k=1}^{n} I_{i,j,k} a_k) = \sum_{k=1}^{n} E(I_{i,j,k} a_k), \text{by linearity of expectation} \\
&= \sum_{k=1}^{n} a_k E(I_{i,j,k}) \\
&\leq \frac{\epsilon}{e} \sum_{k=1}^{n} a_k \\
&= \frac{\epsilon}{e} \|a\|_1
\end{aligned}
$$

Finally we prove that the $Pr[\hat{a_i} > a_i + \epsilon\|a\|_1]$ is bounded by δ, which would imply that $Pr[\hat{a_i} \leq a_i + \epsilon\|a\|_1]$ is at least $1 - \delta$.

$$
\begin{aligned}
Pr[\hat{a_i} > a_i + \epsilon\|a\|_1] &= Pr[\forall_j count[j, h_j(i)] > a_i + \epsilon\|a\|_1] \\
&= Pr[\forall_j a_i + X_{i,j} > a_i + \epsilon\|a\|_1] \\
&= Pr[\forall_j X_{i,j} > eE(X_{i,j})] \\
&= Pr[X_{i,1} > eE(X_{i,1})] \times \ldots \times Pr[X_{i,d} > eE(X_{i,d})] \\
&\text{(due to each hash function being chosen indepdendently from the family)} \\
&\leq e^{-d} \leq \delta, \text{by Markov Inequality}
\end{aligned}
$$

□

3.2 Range Query

A simple approach to answer this query is to compute $\hat{a_i}, \forall i \in [l, r]$ and then returning $\sum_{i=l}^{r} \hat{a_i}$ as the approximate answer. Computing each $\hat{a_i}$ is essentially a *point query*. However, this is not an efficient approach because of the following two reasons: (a) The query time is high, since $(r - l + 1)$ point queries have to be issued; (b) The error guarantee increases linearly with the length of the range, i.e., if R and $\hat{R}$ are the exact and the approximate estimate to the query, then $\hat{R} \leq R + (r - l + 1)\epsilon\|a\|_1$.

Next, we outline an approach to replace the $(r-l+1)$ term with $2\log_2 n$, i.e., $\hat{R} \leq R + 2\epsilon \log n \|a\|_1$. In the following discussion each item a_i is represented by i for ease of notation. Consider the following $\log_2 n$ partitions of the set $\{1, 2, \ldots, n\}$.

$$
\begin{aligned}
\mathcal{P}_0 &= \{1,2,3,4,5,6,7,8,\ldots,n\} \\
\mathcal{P}_1 &= \{\{1,2\},\{3,4\},\{5,6\},\{7,8\},\ldots\} \\
\mathcal{P}_2 &= \{\{1,2,3,4\},\{5,6,7,8\},\ldots\} \\
\mathcal{P}_3 &= \{\{1,2,3,4,5,6,7,8\},\ldots\} \\
&\vdots \\
\mathcal{P}_{\log n} &= \{\{1,2,3,4,5,6,7,8,\ldots,n\}\}
\end{aligned}
$$

The key observation is that any interval $[l,r]$ can be broken into the union of $\alpha \leq 2\log_2 n$ of the above intervals $I_1, I_2, \ldots, I_\alpha$ (known as *dyadic ranges*) s.t.:

1. $I_1 \cup I_2 \cup \ldots \cup I_\alpha = [l,r]$.
2. $I_i \cap I_j = \emptyset$.

For e.g., if $n = 8$ then $[1,7] = [\{1,2,3,4\}] \cup [\{5,6\}] \cup [\{7\}]$. $\{1,2,3,4\}$ comes from $\mathcal{P}_2$, $\{5,6\}$ comes from $\mathcal{P}_1$ and $\{7\}$ comes from $\mathcal{P}_0$. In a general case, no more than 2 intervals will be selected from any $\mathcal{P}_i$.

Now we are ready to explain the efficient technique for answering the range queries. In each $\mathcal{P}_j$, treat each interval as an item. Now build a Count-Min sketch on the items in each $\mathcal{P}_j$. When an update (i_t, c_t) arrives, then for each $\mathcal{P}_j$, the item containing i_t is incremented by c_t. Given a query $[l,r]$, we select α dyadic intervals. For each dyadic interval, ask a point query on the Count-Min sketch corresponding to it. We report the sum of the result of these point queries as the estimate $\hat{R}$. By extending the analysis shown in Lemma 2, it can easily seen that with probability at least $1-\delta$, $\hat{R} \leq R + 2\epsilon \log n \|a\|_1$.

Lemma 3. *The space occupied by the data structure is* $O(\frac{\log n}{\epsilon} \log \frac{1}{\delta})$. *The time taken to update the data structure or to answer the range query is* $O(\log n \log \frac{1}{\delta})$. $R \leq \hat{R}$ *and with probability at least* $1-\delta$, $\hat{R} \leq R + 2\epsilon \log n \|a\|_1$, *where* R *and* $\hat{R}$ *are the exact and the approximate estimate.*

3.3 Heavy Hitters

The naive way to identify the heavy hitters is to exhaustively ask a point query with each $i \in [1,n]$. By Theorem 2 we are guaranteed that all the items with frequency $\geq \phi\|a\|_1$ will be reported, since if $a_i \geq \phi\|a\|_1$, then $\hat{a_i} \geq \phi\|a\|_1$. However, the time to answer the query is too high and impractical.

In the *cash-register model*, where the frequencies only increase, a simple solution can be obtained. Note that an item can start being a heavy hitter following an arrival of that item. So the current set of heavy hitters can be maintained in a heap sorted by the estimated frequency. Also, a count-min sketch is maintained. When the frequency of an item increases, at the same time the sketch can be queried to obtain the current estimated frequency. If the item exceeds the current threshold for being a heavy hitter, it can be added to the data structure. At any time, the current set of approximate heavy hitters will stored in the heap.

In the *turnstile model* (where the value of each item never falls below zero), it is easy to observe that the above approach will not work. A divide and conquer approach is employed to attack this problem: Imagine a binary tree structure on the domain $[1, n]$. Each internal node corresponds to the set of items in the domain which are in the leaves of the subtree of that node. Treat each internal node as a new item and its frequency is the sum of the frequencies of its two children. At each level of the binary tree, we shall construct a count-min sketch based on all the items at that level. (Note that this is similar to the construction used for performing range queries.)

The crucial observation is that if the item corresponding to a leaf node is a heavy hitter, then all the items at its ancestor nodes will also be a heavy hitter in their respective count-min sketch. This leads us to the following simple algorithm: Starting from the root, we only *visit* those nodes which are heavy hitters in their respective sketch. Assume we visited a particular node v. Then we find the frequency of its two children in the sketch they belong to. If any of them also happen to be a heavy hitter, then we visit them recursively till we reach the leaf nodes containing the items which are heavy hitters.

The number of true heavy hitters whose frequency exceeds $\phi\|a\|_1$ is bounded by $1/\phi$. Assuming that not too many false positives are visited by the above algorithm, the number of point queries issues will be $O(\log n/\phi)$.

4 Applications

The count-min sketch is a technique which came from the theoretical computer science community and has found wide range of applications in various domains such as Databases, Natural Language Processing (NLP), Machine Learning, Networking, Data Mining etc. The following are some of the applications mentioned in [3]:

1. Suppose a popular website wants to keep track of statistics on the queries used to search the site. Maintaining the entire log of queries becomes expensive and infeasible (in terms of storage space and query processing) after a point of time. Also, in such applications exact statistics are not necessarily needed, slight approximation is acceptable. A count-min sketch is a useful data structure to maintain in such a scenario.

2. In an online retailer scenario, the items are represented by goods for sale and the frequency is the number of purchases of each item. For an online retailer which performs millions of sales each day, maintaining exact statistics is prohibitive and hence, a count-min sketch would be very useful.

3. To make password guessing difficult, one can keep track of the frequency of passwords online and disallow the ones which are currently popular. This involves keeping an approximate frequency of each password. The paper [3] mentions that the count-min data structure was put into practice to keep track of the frequencies.

5 Final Comments

Cormode and Muthukrishnan introduced the count-min sketch [1]. A detailed discussion on count-min sketch and other related sketches can be found in [2]. To see implementation of count-min sketch in various applications, see [3]. There is a useful website which maintains a catalog of discussions on count-min sketch and its applications [4].

References

[1] Graham Cormode, S. Muthukrishnan: An improved data stream summary: the count-min sketch and its applications. J. Algorithms 55(1): 58-75 (2005)

[2] Graham Cormode: Sketch Techniques for Approximate Query Processing.

[3] Graham Cormode, S. Muthukrishnan: Approximating Data with the Count-Min Sketch. IEEE Software 29(1): 64-69 (2012).

[4] https://sites.google.com/site/countminsketch/