CS369G: Algorithmic Techniques for Big Data Spring 2015-2016

Lecture 7: Heavy Hitters: the Count-Min Sketch

Prof. Moses Charikar *Scribes: Nicole Wein*

1 Overview

This lecture introduces the count-min sketch, a data structure that solves the *heavy hitters problem*:

Given a data stream of length m and a parameter k, find all elements that occur at least m/k times.

2 A non-sketch solution to heavy hitters

The following algorithm is due to Misra and Gries '82 [1].

Algorithm 1

```
initialize $k$ bins, each with an element (initially null) and a counter (initially 0)
for each element $e$ in the stream do
    if $e$ is in a bin $b$ then
        increment $b$'s counter
    elseif find a bin whose counter is 0
        set its element it $e$ and its counter to 1
    else decrement the counter of every bin.
for each bin $b$ do
    $i \leftarrow$ the element in bin $b$
    return $\tilde{f}_i = b$'s counter
```

For all i, let f_i be the true frequency of element i. For each heavy hitter i, the above algorithm returns $\tilde{f}_i$ such that $f_i - m/k \leq \tilde{f}_i \leq f_i$. The proof of correctness is straightforward and will not be included here.

The above algorithm solves the heavy hitters problem efficiently and with little space, but it isn't a sketch so it cannot be used to easily combine information from multiple data streams. At the expense of some space, we can devise a sketch that solves the heavy hitters problem, the count-min sketch.

3 The count-min sketch

The count-min sketch is a solution to the heavy hitters problem developed by Cormode and Muthukrishnan '05 [2]. The idea of the count-min sketch is to use a collection of independent hash functions to hash each element in the stream, keeping track of the number of times each bucket is hashed to. The hope is that for each heavy hitter, one of the buckets it hashes to witnesses few hashes from other elements. The details are as follows.

3.1 The Count-Min Algorithm

Initialization:
- Initialize d independent hash functions $h_j : [n] \to [w]$ with w buckets each for $j \in [d]$. For each bucket b of each hash function j, store a counter $C_j(b)$ initially set to 0.
Building the data structure:
- For each element i of the stream, hash i using each hash function and increment $C(h_j(i))$ $\forall j \in [d]$.
Querying the data structure:
- Given an element i, return $\hat{f}_i = \min_{j \in [d]} C(h_j(i))$.

3.2 Analysis

Let m be the length of the stream. Fix an element i and a hash function h_j. Assume that $h_j(i) = b$. Let's compute the expectation of the value of the bucket b that i hashes to using the j-th hash function:

$$\mathbb{E}[C(h_j(i))] = \mathbb{E}\left[\sum_{s:h_j(s)=b} f_s\right] = f_i + \frac{1}{w}\sum_{s \neq j} f_j \leq f_i + \frac{m}{w}$$

since the sum of all frequencies is just m, the number of elements in the stream, and each element has probability $1/w$ of mapping to a particular bucket. Since the count-min sketch only overestimates frequencies (i.e. $C(h_j(i)) - f_i \geq 0$), we may apply Markov's inequality in conjunction with the above inequality to get:

$$\mathbb{P}\left(C(h_j(i)) \geq f_i + \frac{2m}{w}\right) = \mathbb{P}\left(C(h_j(i)) - f_i \geq \frac{2m}{w}\right) \leq \frac{(\mathbb{E}[C(h_j(i))] - f_i)}{2\frac{m}{w}} \leq \frac{1}{2}$$

Since, we select each hash function $j \in [d]$ independently, we have that:

$$\mathbb{P}\left(\hat{f}_i \geq f_i + \frac{2m}{w}\right) = \prod_{j \in [d]} \mathbb{P}\left(C(h_j(i)) \geq f_i + \frac{2m}{w}\right) \leq \left(\frac{1}{2}\right)^d$$

Choosing $w = \frac{2}{\epsilon}$ and $d = \log_2 \frac{1}{\delta}$ yields $\mathbb{P}\left(\hat{f}_i \geq f_i + \epsilon m\right) \leq \delta$. The space usage is $dw = \frac{2}{\epsilon}\log_2 \frac{1}{\delta}$ counters, plus the space used to store the d hash functions $h_j : [n] \to [m]$. The processing time and query time for a single element are each $d = \log_2 \frac{1}{\delta}$.

4 An extension to the count-min sketch

The count-min sketch does not efficiently support two types of queries that may be useful:

1. range queries i.e. "how many elements in the stream have value between a and b?" (to get this information each element in the range would need to be queried individually)

2. listing all of the heavy hitters (to get this information each element would need to be queried individually)

The following extension to the count-min sketch supports these two types of queries at the expense of some space.

Let n be the number of unique elements. If the elements have a meaningful order, for all i from 1 to n let element i be the i^{th} smallest element. Otherwise, arbitrarily define a bijection between the set of unique elements and the integers 1 through n (in this case range queries are uninformative).

Instead of storing an estimate of the frequency of each individual element, we store an estimate of the total frequency of the elements in certain intervals. The intervals we'll use, shown in figure 1, are known as the *dyadic intervals* of $[1, \ldots, n]$. Formally, the dyadic intervals of $[1, \ldots, n]$ are the set of intervals of the form $[j\frac{n}{2^i}+1, \ldots, (j+1)\frac{n}{2^i}]$ for all i from 0 to $\log_2 n$ and all j from 0 to $2^i - 1$.

For each row of intervals in figure 1, we store a separate count-min sketch data structure. For all rows i, the i^{th} count-min sketch treats two elements that fall into the same interval in row i as the same element.

For all intervals i, let $C(i)$ denote the value that the appropriate count-min sketch returns given i as input.

$$\begin{array}{lcccr}
[1, & & \ldots, & & n] \\
[1, & \ldots, & \frac{n}{2}][\frac{n}{2}+1, & \ldots, & n] \\
& \vdots & & \vdots & \\
[1,2,3,4\ ][\ 5,6,7,8] & & \ldots & & [n-3, n-2, n-1, n] \\
[1,2]\,[3,4][5,6]\,[7,8] & & \ldots & & [n-1, n] \\
[1][2][3][4][5][6][7][8] & & \ldots & & [n]
\end{array}$$

Figure 1. the dyadic intervals of $[1, \ldots, n]$

4.1 Range queries

The following fact will be useful:

Fact: any range in the interval from 1 to n is expressible as the disjoint union of $2 \log n$ intervals in the set of dyadic intervals.

Queries: to perform a range query, simply express the range as a disjoint union I of intervals as in the above fact and return $\sum_{i \in I} C_i$.

Analysis: Let m be the length of the stream. Since, we express our estimate as a sum over $2 \log n$ elements, to achieve an additive error of at most ϵm as in the original count-min sketch, we allow each interval in I to contribute an additive error of $\frac{\epsilon m}{2 \log n}$. Now, the number of hash buckets in each count-min sketch must be $\frac{4 \log n}{\epsilon}$ and there are $\log_2 n$ separate count-min sketches, so the total space needed is $O(\frac{1}{\epsilon} \log \frac{1}{\delta} \log^2 n)$. Each range query requires $2 \log n$ count-min sketch queries for a total query time of $O(\epsilon^{-1} \log \frac{1}{\delta} \log n)$.

4.2 Listing heavy hitters

Let the *frequency of i* denote the sum of the frequencies over all elements in interval i.

It will be useful to view the set of dyadic intervals as a binary tree where the interval in the top row in figure 1 is the root, its children are the two intervals in the 2^{nd} row, their children are the four intervals in the 3^{rd} row, etc. The idea is to explore the tree starting at the root, only exploring the children of intervals of high frequency. Specifically, since the frequency of an interval is at least that of each of its children and the count-min sketch only overestimates frequencies, we need only explore the children of intervals whose frequency, as reported by the count-min sketch, is at least m/k. Starting from the root, we explore the tree in this manner and then return the leaves whose frequency, as reported by the count-min-sketch, is at least m/k.

Analysis: Since $\log_2 n$ count-min sketches are needed, the space requirement is $O(\frac{1}{\epsilon}\log\frac{1}{\delta}\log n)$. (Unlike for range queries, each frequency reported is the result of a single count-min sketch query so another $\log n$ factor is not required.)

Notice that for any given row, the sum over all frequencies in that row is m, the length of the stream. Thus, in any row, there are at most k intervals with frequency m/k. Therefore, we only explore the children of at most k intervals in any given row, so the total number of intervals queried (and the total time required) is $O(k \log n)$.

In the next lecture, we'll introduce the count sketch data structure, which came chronologically before the count-min sketch, but has better performance for certain classes of input.

References

[1] Misra, Jayadev, and David Gries. "Finding repeated elements." Science of computer programming 2.2 (1982): 143-152.

[2] Cormode, Graham, and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." Journal of Algorithms 55.1 (2005): 58-75.