COMP 480/580 — Probabilistic Algorithms and Data Structure Feb 11, 2020

Lecture 9

Lecturer: Anshumali Shrivastava *Scribe By: Jian Huang*

1 Background

Here are some problems that large companies trying to solve:

- Google wants to know what queries are more frequent today than yesterday.
- Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour.
- Social medias like Twitter or Facebook wants to know the trending topics on their websites.

All these problems are hard to solve, because input data elements are entering one after another (i.e., in a stream). It is impossible for any server or database to store the entire stream. Therefore, we need a way to make critical calculations about the stream using a limited amount of memory.

2 Problem Description

The problems mentioned above can all be categorized as Heavy Hitter Problem. It is actually quite common in real life. For example:

- In a twitter feed, what are the top-50 phrases say up to four words in the last year?
- What are popular page views of products on amazon.com given a variety of constraints.
- Stock trends (co-occurrence in sets)

Basically, we need to find the data in the stream that has most hits and approximately how many hits it received.

3 Solutions

Before we start, we need to realize that there is no algorithm that solves the Heavy Hitters problems (for all inputs) in one pass while using a sub-linear amount of auxiliary space.

However, sometimes, we can do better.

3.1 Find Majority Element in Specific Input

If we add some limitations on the input, format the input in the following way:

Given an array A of length n as the input, with the promise that it has a majority element: a value that is repeated in strictly more than n/2 of the array's entries. Your task is to find the majority element.

This can be solved with O(n) running time and O(1) space. Here is the solution:

```
For i = 0 to n-1{
if i == 0
    { current = A[i];
      currentcount = 1;}
else
    {
      if (current == A[i])
          currentcount++
      else
      {
          currentcount - -
          if(currentcount == 0)
              current = A[i]
          }
      }
  }
```

The correctness of the algorithm can be proved with induction.

Hypothesis: given an array A of length n as input, and character a appears more than $n/2$ times in the array, the algorithm will return a with $currentCount$ = number of a $-$ number of other characters.

When $n = 1$, it is obvious that this is true.

Assume it is true for $n = k$, with current count c and majority letter a
When we have the $(K+1)_{th}$ element, if the new letter is a, than the algorithm will still return a with $currentCount = c + 1$, the hypothesis is true.

If the new letter is not a and makes c reduce from 1 to 0, notice that the value of $currentCount$ is still number of a $-$ number of other characters. Therefore in this case, a is not the majority, contradict to our restriction.

If the new letter is not a and $c - 1 > 0$, number of a $-$ number of other characters > 0, a is still the majority.
Therefore, the algorithm is correct.

3.2 Power Law

In real world, most things appear rarely. Only few heavy hitters will appear most of times.

3.3 Bloom Filter With Counter

Bloom Filter can also be used to solve this kind of problems. We only need to make a slight change on the Bloom Filter: instead of only flip the bit in the bit array from 0 to 1, we add 1 to the number at the hash value position. In this way, each entry in the array acts as a counter.

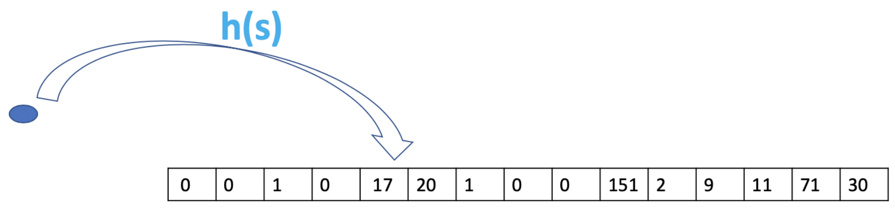

Figure 1: Bloom Filter With Counter

However, there is one situation that we needs to be careful with. If two heavy hitters collide at the same entry in the array, it is very bad for our estimation.

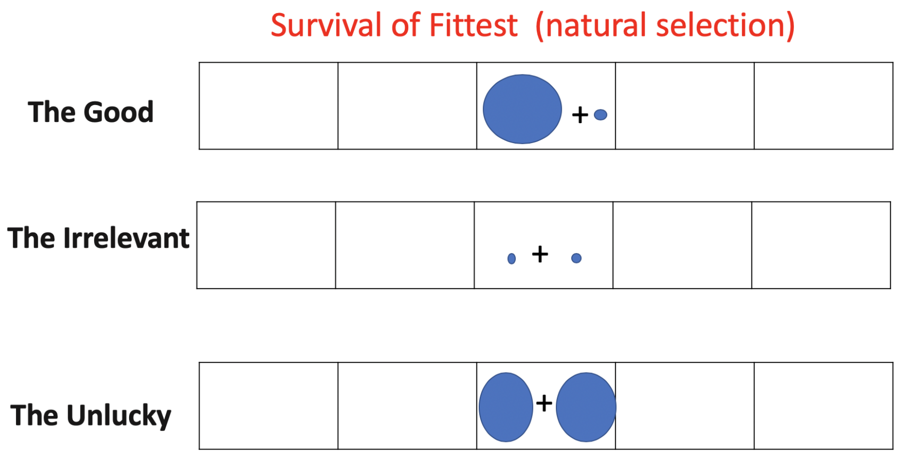

Figure 2: Bloom Filter Situations

3.4 Analysis On Bloom Filter With Counter

We first need to notice:

- If there is no collision, we estimate exactly.
- Collision always over estimate. If we are unlucky, like the situation in Figure 2, we over estimate collision a lot.

Let's say C_s = count of string s.
$\Rightarrow$ If s is frequent (heavy), then h(s) is a good estimate unless some other heavy r collides with h, i.e. h(s) = h(r), or too many tails present in h(s)

Let's say Σ =sum of all elements = $\Sigma_s C_s$, R = Range of the array in Bloom Filter.
$\Rightarrow$ For each element s, every other element have $\frac{1}{R}$ probability to collide with s
$\Rightarrow$ Expected value of counter $h(s) = C_s + \frac{1}{R}(\Sigma - C_s) < C_s + \frac{\Sigma}{R}$
$\Rightarrow$ Expected Error $< \frac{\Sigma}{R}$

Also, in power law, if s is head then $C_s = f\Sigma$ (some fraction of everything)

Our goal is to make the probability of unlucky situation smaller. This can be done through **Count-Min Sketch**

4 Count-Min Sketch

4.1 Algorithm

The idea is actually simple: We do Bloom Filter with counter independently for multiple times. For the query of a string, we return the minimum count among all the Bloom Filter Arrays. (Notice here each array corresponds to a distinct hash function)

Here is the summary of the algorithm:

- Take d independent hash function $h_1, ..h_d$ and d arrays of size R.
- **Update**: Any string s, increment $h_i(s)$ in array i.
- **Estimate Counts**: Given query q, return $min(h_1(q), h_2(q), ..., h_d(q)$.
- **Update and Estimate cost O(d)**.

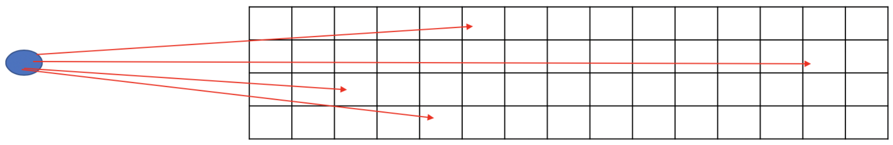

Figure 3: Count-Min Method

This brings us two advantages:

- Unless all the d counters messes up simultaneously, we have a good estimate.
- For heavy hitters, unless every d counters has more than 2 heavy hitters we are fine.

So how good it is? How do the values of d affect the error? What is the bound of probability of large error? How much memory do we need? We will answer these questions in the analysis session.

4.2 Analysis

We define the unlucky situation as the error is larger than 2 times the expected error rate. We will use Markov inequality to analyze the probability that this happens.

Markov Inequality: $Pr(X > a) < \frac{E(X)}{a}$; $X \geq 0$
For just 1 hash function, let's say $\frac{1}{R} = \epsilon$
$\Rightarrow$ Expected error (err) $= \frac{\Sigma}{R} = \epsilon\Sigma$
$\Rightarrow Pr(err > 2\epsilon\Sigma) < \frac{E(err)}{2\epsilon\Sigma} = \frac{1}{2}$

If the error is worse than $2\epsilon\Sigma$, then all the d hash functions must result in unlucky situation. The probability that one hash function lands into unlucky situation is less than $\frac{1}{2}$, therefore with d hash functions, the probability is bounded by 0.5^d

Based on the calculation above, we can see that if we want to decrease the error, we need to increase R. If we want to decrease the probability that large error happens, we need to increase d.

Now let's see how much memory we need to manage the probability of bad errors.
Let's say the expected count for string s is $\widehat{C_s}$. We want to ensure the probability $C_s < \widehat{C_s} < C_s + 2\epsilon\Sigma$ is $1 - \delta$ for any given string s. In other words, the probability of large error for each string is δ.
Since the probability of large error is bounded by 0.5^d, we have:
$\Rightarrow 0.5^d < \delta$
$\Rightarrow d * \log 0.5 < \log \delta$
$\Rightarrow d < \frac{\log \delta}{\log 0.5} = -\log \delta = \log \frac{1}{\delta}$
The range of each array is $R = \frac{1}{\epsilon}$
Therefore, for the probability of $C_s < \widehat{C_s} < C_s + 2\epsilon\Sigma$ is $1 - \delta$ for a given string, we need $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ memory space.
However, This is for one string. If the probability of error appear in a given string is δ, then the probability that it appears on any one of the N strings is $N * \delta$. To make the probability of large error for each string is still smaller than δ, we need to use $\delta' = \frac{N}{\delta}$, and therefore we need $O(\frac{1}{\epsilon} \log \frac{1}{\delta'}) = O(\frac{1}{\epsilon} \log \frac{N}{\delta})$ memory.

4.3 Identify Top K

What if we want to identify the top K heavy hitters? It turns out that we can use a minheap of size K. Whenever, we see a string s, update, estimate and check if the estimate is less than min of top-k heap, update heap if more. $(O(log(k)))$. The total update is $O(d \log K)$ in worst case.

References

Lecture 9 slides available on https://www.cs.rice.edu/ as143/COMP480_580_Spring20/index.html