COMP 480/580 — Probabilistic Algorithms and Data Structure Sep 20, 2022

Lecture 9 – Stream Estimation 1: Count-Min Sketch

Lecturer: Anshumali Shrivastava
Scribe By: Manueal Diaz, Anthony Cai, and Jefferson Hernandez

1 Heavy Hitters problem

In many applications of interest, we wish to find items that occur more than one would expect. These might be called outliers or anomalies, or any other statistical terms. This idea is captured by the heavy hitters problem—we suppose that there are only a few large or "heavy hitting" elements in the stream, and we want to have a streaming algorithm that can identify them.

1.1 Motivating example (Google Trends)

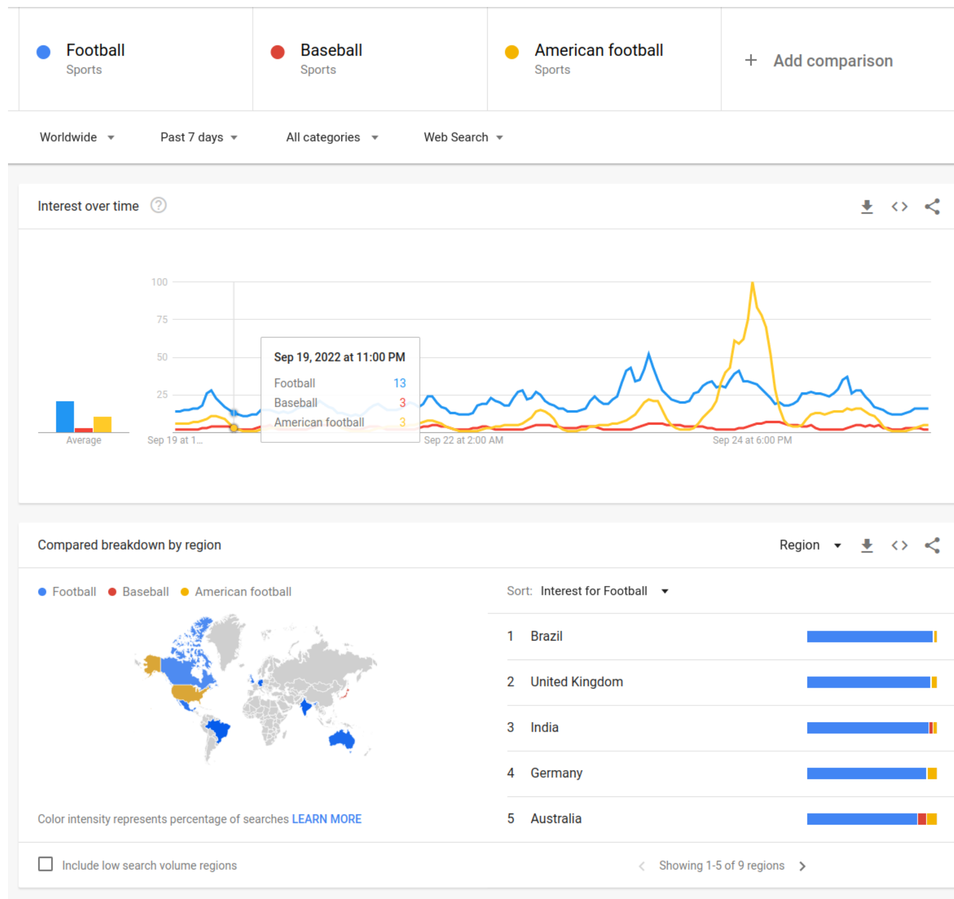

Figure 1: Google Trend search for the terms "Football", "Baseball", and "American Football"

Google Trends. is a page by Google that tracks surgeon search queries around the world in real time. Its goal is not only to serve queries but to serve queries fast. Google is able to do this

by keeping track of so-called "heavy hitters"—search terms that make up a disproportionate amount of traffic—Google can serve these faster if there were a way to cache them. On a normal day, Google would process approximately 7 Billion searches if not more. It becomes obvious that, given the diversity in queries and their sheer amount, we can not store them. In more general terms, for these kinds of problems it becomes impossible to maintain a data structure with size proportional to the input size, somehow we must come up with a data structure that takes *sublinear* space.

2 Streaming

Streams are one of the most common ways that data is transmitted and are extremely similar to how networks handle data. They are extremely difficult to work with however since you do not know when the stream will end, and due to the large amount of data that can be transmitted. In some cases it is possible for the amount of data being streamed to be in petabytes.

A stream is a model of computation that emphasizes *space* over all else. You only get to look at the data once, or sometimes you can make two passes over the data. More formally, we can define the stream as the sequence $A = \langle a_1, a_2, \cdots, a_n \rangle$ of m items where each $a_i \in [m]$, sot it takes $\log(m)$ to represent each a_i. Notice that just counting the elements requires $\log(n)$ space. The goal of a streaming algorithm is to compute a quantity $g(A)$. From now on let $c_i = |\{a_i \in A, \quad a_i = j\}|$ represent the number of items in the stream that have value j.

3 Majority and Heavy Hitters

One of the most common heavy hitters problem is that of finding the majority element. Consider the array A of size m, you are told that it has a majority element, an element whose count $c_j > n/2$, the goal is find such j. Note that a majority element is not necessarily guaranteed to exist. Also note that if the majority element does exist, it is unique.

Algorithm 1 Majority(A)

Require: $c = 0$ and $l = \emptyset$
  for $i = 1$ to m **do**
    if $(a_i = l)$ **then**
      $c = c + 1$
    else
      $c = c - 1$
    end if
    if $(c \leq 0)$ **then**
      $c = 1, l = a_i$
    end if
  end for
  return l

Notice that this algorithm is correct just see that if the majority element exists:

- If $(l = j)$, then c can be decremented at most $< n/2$, but $c > n/2$.
- If $(l \neq j)$, then c can be decremented, but incremented $> n/2$

But if the majority element does not exist, any answer is OK.

We can formalize the heavy hitters problem, as follows, from an array A with size n, compute all elements that occur at least n/k times for some reasonable k [1]. Notice that finding the majority element corresponds to setting $k = 1 - \delta$, for a small value $\delta > 0$.

3.1 An Impossibility Result

Definition 1 *There is no algorithm that solves the Heavy Hitters problems (for all inputs) in one pass while using a sublinear amount of auxiliary space.*

To see so, suppose that we set $k = n/2$, meaning we want to find all elements that appear at least twice in the array. Suppose that the stream, we has seen so far contains the elements $x_1, x_2, \cdots, x_n$, where each x_i is different, each time we get a new elements y, our problems reduces to determining whether $y \in A$?. (Notice that the simplest way to solve this is using a hash table). We can easily see that we cannot answer each query "is $y \in A$?" without storing all previous elements and thus using linear space. So if we throw out some elements, it becomes impossible to answer queries for elements we already throw out!! Thus giving an intuition of why solving the heavy hitter problems is very difficult. A more formal proof can be made for all k using the Pigeonhole Principle.

3.2 Can we do better?

As we saw, the size of out data structure is bound to grow with the size of the stream, and it is not always possible to beat a naive algorithm without some data-dependent assumptions. That means we need to relax the problem. that relaxation is called ϵ-approximate heavy hitters (ϵ-HH) presented in [4]. We require an array A of size n and user-defined parameter k and ϵ. This algorithm will produce a data structure that follows the following rules:

1. Every value that occurs at least n/k times in A is in the list.
2. Every value in the list occurs at least $n/k - \epsilon n$ times in A.

Notice that as $\epsilon \to 0$, then the ϵ-HH is equivalent to the heavy hitter problem. This means that the space occupied by the algorithm grows by $\mathcal{O}(\frac{1}{\epsilon})$. Finally, an algorithm that follow conditions (1) and (2) is a useful as the original heavy hitter problems but with much less space consumption. One of such data structures, which is also elegantly simple, is called the *count-min sketch* [1]. We remark to the reader that there are other structures that are based on Algorithm 1 such as [2, 3] with the caveat that they are deterministic in nature or much more complex.

4 Count-Min Sketch

The count-min sketch is a data structure similar to the bloom filter in that it uses d hash functions and arrays of size R. The count-min sketch support two operations an increment operation `Inc(x)` and a count operation `Count(x)` [2]. The operation `Count(x)` if supposed to return the count c_x, in other words, how many times we have called `Inc(x)`.

[1] You must think of n in the billions and k in the 1000s or 10000s.

[2] Notice that the same data structure supports bigger than one increments in that case you would define $\texttt{Inc}(\texttt{x}, \Delta)$ for $\Delta \geq 0$ Likewise, it can support deletions but we focus on the case where increments are $+1$ for simplicity.

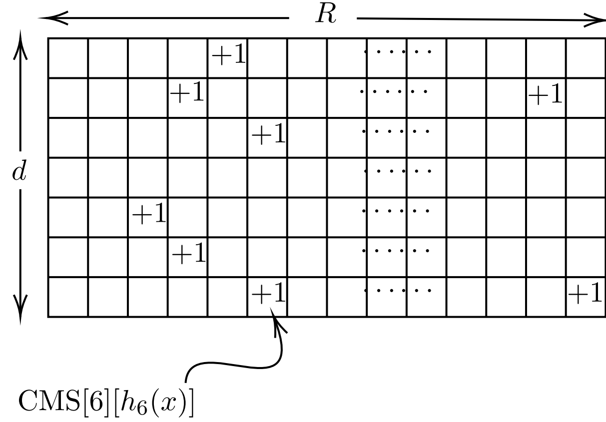

Figure 2: The result of applying `Inc(x)` on the CMS data structure. Each row corresponds to a hash function.

The count-min sketch has two parameters the number of hash functions d and the number of buckets R. See Figure 2. The objective of the data structure is to compress the data, since $R \ll n$, but since this compression will come with errors (hash collisions), we need to add independent trials (remember power of two choices) using d hash functions.

The final structure will be an array of $d \times R$ CMS counters each initialized at 0. The code to apply the `Inc` operation is as follows

Algorithm 2 `Inc` Function

Require: $h_1, h_2, ..., h_d$ Hash functions, $d \times R$ CMS array, and x value to be hashed
  for $i = 1$ to d **do**
    $\text{CMS}[i][h_i(x)] + +$
  end for

To motivate the implementation of `Count`, notice that the biggest downside is that since you are hashing to a range that is smaller than the total amount of unique data points, there will be collisions. Since counters are never decremented this means that the total count for any data point will be over what the true value is.

$$\text{CMS}[i][h_i(x)] \geq c_x, \tag{1}$$

where c_x denotes the true count of item x. For a heavy hitters problem, this materializes into three distinct cases:

- Lucky collisions: small counts collide with large counts. This causes us to overestimate the heavy hitters, but only a small amount
- Irrelevant: small counts collide with other small counts. This gives a bad estimate of small counts, but we are only interested in heavy hitters.
- Unlucky: elements with large counts collide with other elements that have large count. This causes us to dramatically overestimate the count for each element.

We remark the count estimate $\text{CMS}[i][h_i(x)]$ can not underestimate the true count c_x, but it generally overestimates it.

Algorithm 3 `Count` Function

Require: $h_1, h_2, ..., h_d$ Hash functions, $d \times R$ CMS array, and x value to be hashed.
  return $\min\left(\text{CMS}[1][h_1(x)], \text{CMS}[2][h_2(x)], \cdots, \text{CMS}[d][h_d(x)]\right)$

Notice that all error con the count-min sketch are one sided, one natural question to ask is *How large are the errors?*. This will also allow us to have better intuition on how to set d and R. But first let us see how this data structure relates to bloom filters.

4.1 Relation to the bloom filter

Hashing into a counter almost provided a way to keep track of how often things were seen, but the chance of unfortunate collisions keeps it from being a valid solution. The simplest way to achieve this, is to simply add more hashing functions and have an array for each function. This is effectively creating a bloom filter that maps to counters. Where as a traditional bloom filter will minimize the amount of memory needed, this version will minimize the over estimate. Here we found a correspondence between the over-estimation of the count-min sketch and the bloom filter's property that it only suffers from false positives

4.2 Overestimation of Count-Min Sketch

Let us define the variables c_x which denotes the true count of object x, likewise define $Z_i = \text{CMS}[i][h_i(x)]$ to represent our estimate of the count, and $\sum c_x = \Sigma$. Notice that Z_i is a random variable as long as our hash function are randomly sampled from an universal hash family and independent. Considering the existence of collisions since $R \ll n$, we can write our estimate as follows

$$Z_i = c_x + \sum_{y \in S} c_y, \tag{2}$$

where $S = \{y \neq x : h_i(y) = h_i(x)\}$ denotes the objects that collide with x in the i-th row. Recall that if h_i is at least 2-universal then for every x, y that are distinct $\mathbf{Pr}[h_i(y) = h_i(x)] \leq 1/R$. We can rewrite Equation 2 as follows

$$Z_i = c_x + \sum_{y \neq x} c_y \mathbb{1}_y, \tag{3}$$

where $\mathbb{1}_y$ is the indicator variable that is equal to 1 if $h_i(y) = h_i(x)$ and to 0 otherwise. Applying linearity of expectation we get

$$\mathbb{E}[Z_i] = c_x + \sum_{y \neq x} c_y \mathbb{E}[\mathbb{1}_y], \tag{4}$$

Remember that for the indicator variable, its expectation is $\mathbb{E}[\mathbb{1}_y] = \mathbf{Pr}[h_i(y) = h_i(x)] \leq 1/R$ which let us write

$$\mathbb{E}[Z_i] \leq c_x + \frac{1}{R} \sum_{y \neq x} c_y \leq c_x + \frac{\Sigma}{R}, \tag{5}$$

We can translate this bound from one in expectation to one in probability using a concentration inequality, in this case Markov's. Let us define the quantity

$$\text{Error} = Z_i - c_x \geq 0,$$

as the amount of error made by our estimate of the count c_x. This error has expected value equal to $\mathbb{E}[\text{Error}] = \Sigma/R$, which looks similar to the definition of ϵ-HH and allows to have $\epsilon = 1/R$. We would want that the probability that we overestimate the error by twice this amount be less than 1/2 otherwise we will contradict Equation 5.

$$\begin{aligned} \mathbf{Pr}[\text{Error} > 2\epsilon\Sigma] < \frac{1}{2}, \\ \mathbf{Pr}[Z_i > c_x + \epsilon\Sigma] < \frac{1}{2} \end{aligned} \tag{6}$$

Assuming that the hash functions are chosen independently, we have

$$\mathbf{Pr}[\min_{i=1}^{d} Z_i > c_x + \epsilon\Sigma] = \prod_{i=1}^{d} \mathbf{Pr}[Z_i > c_x + \epsilon\Sigma] < \left(\frac{1}{2}\right)^d, \tag{7}$$

which we can solve for d and obtain $d = \log_2(\frac{1}{\delta})$ for any user specified probability δ. This means that for $\delta = \epsilon = 0.01$ we need 6 or 7 hash functions and an array of barely over a 1000 which does not at all depend on the size of the stream n. We finally remark that the space used by the count-min sketch of $\mathcal{O}(\frac{2}{\epsilon}\log_2(\frac{1}{\delta}))$, and that if we chose simple but universal hash function the time of the operations `Inc` and `Count` is $\mathcal{O}(\log_2(\frac{1}{\delta}))$

5 How to identify Top-k?

When it turns out to be top-K heavy hitters, we need to consider the memory usage as well as using heap structures to keep track on the result. The estimate c_s is guaranteed in the range $c_s < \hat{c_s} < c_s + 2\epsilon\Sigma$. For example, $c_s = f \cdot \Sigma$ if s becomes a heavy hitter. We will need to choose an ideal d such that the error can be limited with $\epsilon < f$. High probability decays as 0.5^d, and thus $d \in 4, 5$ suffice for the general cases.

5.1 Memory Requirement

To manage the probability of unwanted errors, for any given string s the expect count is $\hat{c_s}$ and the probability of $c_s < \hat{c_s} < c_s + 2\epsilon\Sigma$ is 1-δ for any given string s. The time complexity can be expressed as O($\frac{1}{\epsilon}log(\frac{1}{\delta})$):

$$\begin{aligned} 0.5^d &< \delta \\ d \cdot log(0.5) &< log(\delta) \\ d &< \frac{log(\delta)}{log(0.5)} \\ d < -log(\delta) &= log(\frac{1}{\delta}) \end{aligned} \tag{8}$$

If the error probability in a given string is δ, than for N strings we have $N \cdot \delta$. In order to keep the error within δ we choose $\delta = \frac{\delta}{N}$. So in the original notation $O(\frac{1}{\epsilon}\ log(\frac{1}{\delta}))$, $\frac{1}{\delta}$ becomes $\frac{N}{\delta}$ and then we derive the memory requirement $O(\frac{1}{\epsilon}\log(\frac{N}{\delta}))$.The number of strings (N) is complied under logarithm so there is exponentially less space needed.

5.2 Analysis & Turnstile Model of Stream

Furthermore, when we identify Top-K elements Min Heap of size k is used to update, evaluate, and check if the estimation reaches the min of Top-K heap. If so we update the heap with $O(log(k))$. In the worst case, the cost for all hash functions is $d \cdot log(k)$.

An interesting application is the turnstile setting if we assume an N dimensional vector V, $O(N)$ space is unacceptable. Given a time stamp t, we are only able to see (i, δ_i), but by hashing it d times and add δ_i to all has counters we may find a convenient method to handle turnstile streaming as $O(N)$ is common for most general problems, using power law input we can make exponential improvements.

References

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

[2] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*, pages 398–412. Springer, 2005.

[3] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[4] Tim Roughgarden and Gregory Valiant. Cs168: The modern algorithmic toolbox lecture# 2: Approximate heavy hitters and the count-min sketch, 2015.