COMP 480/580 — Probabilistic Algorithms and Data Structure Feb 5, 2019

Lecture 9: Count-Min Sketch

Lecturer: Anshumali Shrivastava *Scribe By: Minh Le*

1 The Heavy Hitters Problem

1.1 Problem Description

In many applications of practical interest, we wish to identify only the "most important" or largest values in a data stream. 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.

We motivate the heavy hitter problem using a twitter data-mining application. On twitter, certain words and phrases are important and are repeated often in many tweets. These phrases are the heavy hitters. We would like to identify these important phrases in a stream of tweets. Since our solution must be a streaming algorithm, we want that the memory and runtime of our algorithm to scale appropriately. More concretely:

- Given a stream of tweets, find the top 50 phrases in the last year
- Each phrase cannot contain more than 4 words

A naive solution is to keep a count for all possible phrases of four words (4-grams)

- Use a dictionary, with keys being all possible combinations (4-grams)
- As we iterate through the stream, increase the value of the count corresponding to the given 4-gram

This solution is correct and exact, but requires a tremendous amount of memory. Suppose that tweets can be assembled from one million words and observe that the number of possible phrases is $(10^6)^4 = 10^{24}$! This memory cost is unrealistic in a practical context. Even if we intelligently choose the 4-gram keys, we still require 1.6TB of memory just to store the count array, let alone the keys themselves. This solution does not scale well.

1.2 HH Applications

The heavy hitters problem is common in practice. Here we provide a few more examples of applications to motivate our discussion.

- Computing popular products, given a particular context: For example, we want to identify popular products on amazon.com given a variety of constraints
- Identifying heavy TCP flows. We are given a list of data packets passing through a network switch, each annotated with a source-destination pair of IP addresses. It is common for a few source-destination pairs (the heavy hitters) to be sending most of the network traffic. We want to identify the heaviest network flows. This is useful for identifying denial-of-service attacks.
- Stock trends (co-occurrence in sets)

1.3 Can we do better?

The main difficulty with the HH problem is that, without data-dependent assumptions, the size of our sketch scales with the number of elements in the stream. It is not always possible to do better than our naive algorithm, since there is no algorithm that solves the Heavy Hitters problem (for all inputs) in one pass while using a sublinear amount of auxiliary space. This impossibility result is easy to prove using the pigeonhole principle. However, if we make some assumptions about the inputs, we can obtain improvements.

2 Majority Element Problem

First, we start with an easier problem - the majority element problem. This example will show how assumptions about our inputs can lead to algorithmic improvements.

Definition 1. Majority Element
Given an array A of length n, the majority element of A is the element that is repeated strictly more than $\frac{n}{2}$ times in A.

Note that a majority element is not necessarily guaranteed to exist. Also note that if the majority element does exist, it it unique.

2.1 Problem statement

Suppose that an array A is guaranteed to contain a majority element. Find the majority element.

2.2 Candidate Solutions

- Solution 1: Sort A and find the median: $O(nlogn)$ (We suppose that a median finding algorithm exists and runs in $O(n)$)
- Solution 2: Recursive solution: Repeatedly split A in half to find the majority element $O(nlogn)$
- Solution 3: One-pass streaming algorithm: $O(n)$, using $O(1)$ storage

Algorithm 1: Solution 3

```
Input: Array $A$ of length $n$
Output: The majority element
1 for $i$ from 0 to $n-1$ do
      if $i == 0$ then
2         $current \leftarrow A_i$;
3         $currentcount \leftarrow 1$;
      else
          if $current == A_i$ then
4             $currentcount++$;
          else
5             $currentcount--$;
6             if $currentcount == 0$ then
7                 $current \leftarrow A_i$;
8                 $currentcount \leftarrow 1$;
9 return $G$
```

A proof sketch is as follows. Solution 3 is correct because if the majority element exists, there will never be a case where the decrement decreases *currentcount* for the majority element to 0. (The max value of *currentcount* for the majority element is more than $n/2$, and we subtract at most $n/2$ from this value). Note that this solution only applies when we *assume that a majority element exists*. In particular, solution 3 cannot determine whether an array contains a majority element.

2.3 Power Law

The solutions to the majority element problem do not apply to the general case, when a majority element is not guaranteed to exist. However, by making assumptions we can reduce the problem to a much more tractable form. We apply the same idea to the heavy hitter problem. We require that most elements in our stream be small, with only a few large (heavy hitter) elements. This assumption is known as the Power Law.

If the Power Law assumption holds for our input, then we can do much better than the naive solution! Fortunately, most real word data follows the Power Law. For instance, a small selection of English phrases are used very frequently on Twitter, but most random phrases do not make sense and are not frequently present in real-world data.

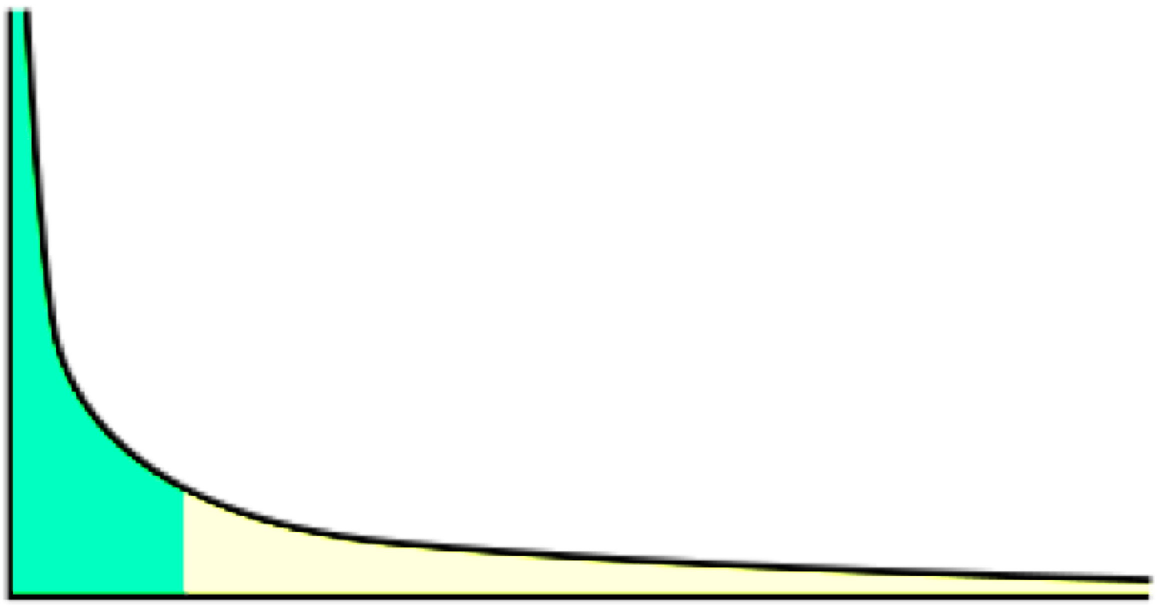

Figure 1: Power law. Suppose we sort the weights of elements in our stream. For the heavy hitters problem, we expect the distribution to fall off faster than x^{-k} for some power k. Here, x-axis shows elements, y axis is contribution/weight of the elements

3 Bloom Filters with Counter

As mentioned in previous lectures, a bloom filter is a bit array that uses k-independent hashing functions. When we hash an item x, the bloom filter simply flips the bit at $h_i(x)$ for all $i \in [1, k]$ to 1. The false positive rate scales with the length of the array.

To solve the HH problem with a power law assumption on the input, we will first examine a modified form of the bloom filter. Instead of flipping bits at $h_i(x)$, our modified bloom filter will contain an array of integer counts. To hash an item, we will increment the count at $h_i(x)$. If element x has a collision with element y, we simply add the counts of x and y (that is, bucket $h_i(x)$ contains the counts of x *and* y). For simplicity, we will only consider the case where $k = 1$. In this case, the probability $Pr(h(x) = c) = 1/R$.

Given a few heavy hitting elements with large counts and many other elements with small counts, there are 4 possible outcomes for our bloom filter.

1. No collisions: We get a perfect estimate of the counts for each element

2. Collisions exist: We always overestimate (see figure for more detail)

   - 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.

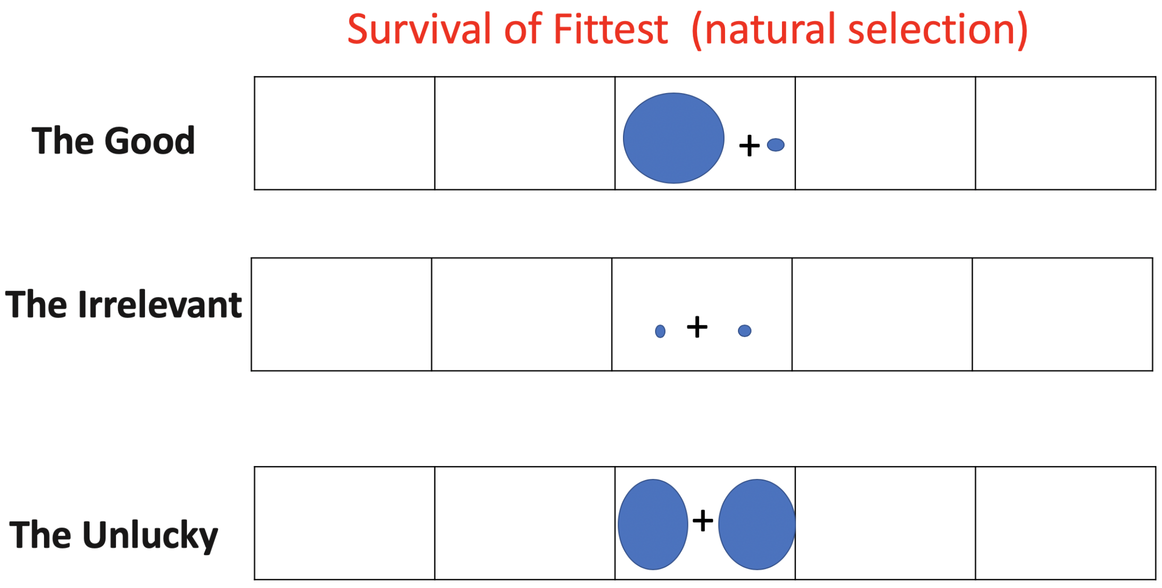

Figure 2: Intuition of bloom filters for the heavy hitter problem. If a small count collides with a large one (The Good), then we still have a good estimate of the large count. If two small counts collide (The Irrelevant), it does not affect our estimate of a heavy hitting element. If two large counts collide (The Unlucky), we have a bad estimate of the large counts.

3.1 Analysis

Now we consider the expected error. Let the count of string s be c_s. 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 if too many infrequent elements collide with $h(s)$. Let the revised bloom filter A have length R, and observe that

$$E[h(s)] = E[A[h(s)]] = E[c_s + \sum_i 1_{\{h(i)=h(s)\}} c_i]$$

where $1_{\{h(i)=h(s)\}}$ is the indicator random variable of a collision happening between i and s. Therefore

$$E[h(s)] = c_s + \sum_{i \neq s} \frac{c_i}{R}$$

We wish to estimate c_s, and so the error $E[h(s)] - c_s$. The expected error of our estimator is

$$E[error] = \sum_{i \neq s} \frac{c_i}{R}$$

To determine this value, let Σ be the summation of all elements ($\Sigma = \sum_s c_s$), then

$$E[h(s)] = c_s + \sum_{i \neq s} \frac{c_i}{R} < c_s + \frac{\Sigma}{R}$$

Thus, $E[error] < \frac{\Sigma}{R}$. Suppose our data obeys the power law. Then if s is a heavy hitter, $c_s = f\Sigma$.

4 Count-min Sketch

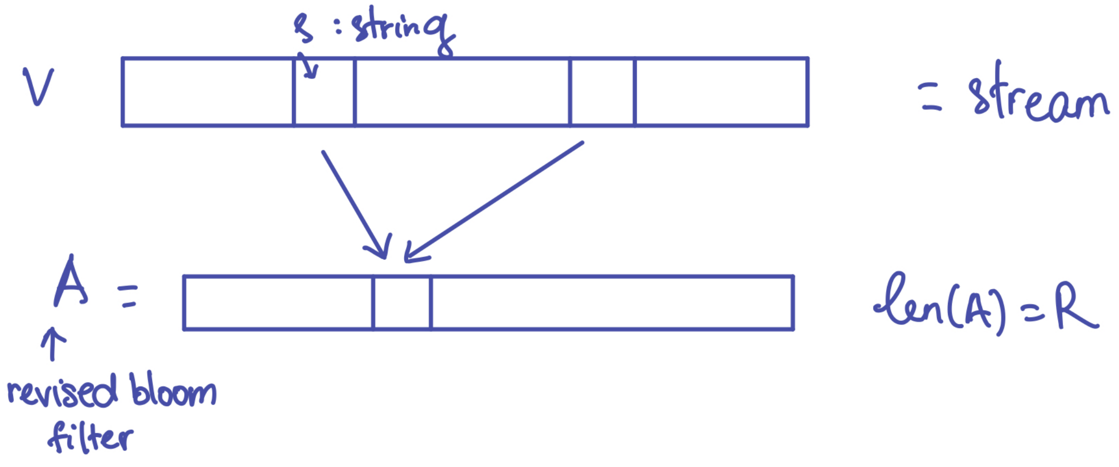

Figure 3: Update process for the modified bloom filter. To form a count min sketch, we repeat the modified bloom filter d times.

4.1 The Algorithm

We can use the idea of a count bloom filter to build an algorithm that correctly estimates the heavy hitter counts with high probability. We will simply repeat the count bloom filter independently d times and take the minimum value (Count-Min). The intuition is that by repeating the count bloom filter, we minimize the likelihood that we are unlucky for all d counters. Unless all d counters are wrong simultaneously, we will have a good estimate.

Algorithm 2: Count-min Sketch Update

```
Input: $d$ Array of length $R$, $d$ independent hash functions, any string $S$
Output: An updated CMS
1 for $i$ from 0 to $d-1$ do
2     $h_i(S) += 1$ in array $i$ ;
3 return The $d$ arrays of length $R$
```

Algorithm 3: Count-min Sketch Estimate (Query)

```
Input: $d$ Array of length $R$, $d$ independent hash functions, query $q$
Output: Estimated count for query $q$
1 return $min(h_0(q), h_1(q), ...h_{d-1}(q)$
```

The update cost for this algorithm is $O(n)$ and the estimation cost is $O(nlogn)$.

4.2 Error rate

If we only use one hash function and let $\epsilon = \frac{1}{R}$, the expected error is

$$E(error) = \epsilon\Sigma$$

Using the Markov Inequality, we have

$$Pr(err > 2\epsilon\Sigma) < 1/2$$

In words, the error is larger than $2\epsilon\Sigma$ when we use d bloom filter repetitions with probability 0.5^d. For example, if we take $d = 7$, the probability is 0.007.

5 Count Min Sketch and Heavy Hitters

The estimate c_s is bounded within the range: $c_s < \hat{c_s} < c_s + 2\epsilon\Sigma$ with very high probability. If s is a heavy hitter then $c_s = f\Sigma$. Therefore, for heavy hitters we want to choose d so that we have ϵ smaller than f and get a relative error guarantee. In practice, $d = 4$ is often sufficient.

5.1 Memory requirement

To identify heavy hitters with probability $1 - \delta$, we require the count min sketch to have the error bound from section 4.2 for all of our n inputs. For each input, we need $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ memory for the bound. To see this, solve for d such that the error bound ϵ holds.

This must be true for all n input strings. Using the union bound, this happens if we pick $\delta = \frac{\delta}{N}$. The total memory requirement is $O(\frac{1}{\epsilon} \log \frac{n}{\delta})$, which is exponentially less space (log memory space).

5.2 How to identify top-k

To identify the top-k elements, we can use a min heap of size k. Whenever we see a string s, we update the sketch, estimate the count and check if the estimate is less than the minimum of our top-k heap. If not, we update the heap to include s. The computational cost to update the min heap is $O(log(k))$, therefore the total update cost is $O(dlogk)$ in the worst case (when we must update the heap).