Lecture 6 (09-14) - Streaming Algorithms and Count Min Sketch

The basic idea of a **streaming model** is that we have access to some stream of ordered data: $a_1, a_2, a_3, a_4 \cdots \in$ an alphabet Σ, and would like to know some properties about it. Unlike with a traditional list or array input, the data stream will be assumed too large to store and must be processed as it comes in.

It may be useful to define a function $F(a_{[1:t]})$ that processes the stream and outputs the property for the stream up to the t^{th} element in the stream.

Examples of Streaming Algorithms

For example, suppose we are given a stream of integers. We can let F compute the sum of the integers in the stream.

Of course, this is quite simple: $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. Likewise, it may be useful to define it as: $F(a_{[1:t]}) = a_t + F(a_{[1:t-1]})$ to give us an idea of how each element of the stream is processed individually.

For streaming algorithms, the space we use is also very important. Of course, it would be trivial to do every streaming algorithm if we simply append the elements to an internal list. We'd like to limit it.

For the summing problem, consider each element of the stream to be b bits. Then the space used to store the max is $O(b + lg(t))$ where the worst-case scenario is a stream of exclusively the maximum element.

Some other algorithms include:

- Finding the maximum (relatively easy) - store the current max and compare each new element to it. Takes $O(b)$ space.
- Finding the median. (harder) - for low space, use ϵn-approximation (will be off by at most $\pm\epsilon n$), taking $O(\frac{1}{\epsilon}\log(\mid \Sigma \mid))$ space. Given $b = lg(\mid \Sigma \mid)$, we get $O\left(\frac{b}{\epsilon}\right)$ space.
- Finding the number of distinct elements

And finally...

The **Heavy Hitters Problem**

There are many variations of this problem, but they all focus on the idea of element frequency. In particular, we want to find elements that appear at least a fraction ϵ within the stream. This may be useful, for instance, if we want to view the most popular websites in a stream of internet traffic.

Of course, given the limitations of streaming, we generally have two options to deal with a difficult problem such as this one:

- We can compute an approximate but not completely correct solution that may be good enough for our purposes (i.e. we allow for some false positive solutions). Some of these compromises may allow for better space efficiency.
- Hashing can be leveraged to provide use with more exact solutions.

More formally defining the problem, suppose each element in our stream is a tuple of an operation (ins or del) and a vector of dimension $\mid \Sigma \mid$. The vector x_i will be a one-hot encoding of which element e we are inserting or deleting. We will abstract our stream state to a single vector E with the following operations:

- Insert: E + x_i
- Delete: $E - x_i$
- Check count of e: look up the i^{th} value that corresponds to the one-hot encoding of e. We will call the output E_i^t for the count of the i^{th} value at time t.

Our job is to maintain a data structure S using as little space as possible that we can perform a lookup on some element $e \in \Sigma$ and check the count. To allow us to solve the problem in smaller space, we are also allowing ourselves an error of $\pm\epsilon||E||_1$, meaning false positives are allowed if are relatively close, where ϵ is some fraction such as $\frac{1}{2}$ that we may be interested in checking if an element has appeared with a frequency of ϵ in the stream.

Lastly, we will assume that we cannot delete elements that do not have at least an equal number of insertions.

The Solution

As previously discussed, we will leverage hashing (specifically universal) in order to solve this problem. Let's maintain an array A of size M that we will hash into. For now, we don't know what M is, but we'll set it to minimize the space we need. Now, we will have a universal hash function H and provide the following functions:

```
insert(i) {
        A[H(i)]++
}

delete(i) {
        A[H(i)]--
}

lookup(e) {
        return A[H(i)]
}
```

Showing Correctness

Clearly in a perfect world where we don't have collisions, we would return the correct count every time; however, we don't live in that world. That's see how much of an error term the collisions actually cause us. Let us call the value returned $\tilde{E}_i$:

- First, we know that $\tilde{E}_i \geq E_i$ as we will count all instances of the i^{th} element itself.
- The rest of the count we get are collisions:

$$\tilde{E}_i = E_i + \sum_{x \neq e} count(x)\mathbb{1}[h(x) = h(e)]$$

  where $\mathbb{1}[]$ is the 0/1 function that is 1 when the statement is true and 0 otherwise.

Because we are using a universal hash function, we can analyze the expected value of this error term.

$$E[\tilde{E}_i] = E[E_i + \sum_{x \neq e} E_x \mathbb{1}[h(x) = h(e)]]$$

$$E[\tilde{E}_i] = E_i + \sum_{x \neq e} E_x E[\mathbb{1}[h(x) = h(e)]]$$

As $\mathbb{1}[h(x) = h(e)]$ is essentially just the random variable we defined earlier, this is just:

$$E[\tilde{E}_i] = E_i + \sum_{x \neq e} E_x P(h(x) = h(e))]$$

$$E[\tilde{E}_i] = E_i + \sum_{x \neq e} \frac{E_x}{M}$$

Then, the sum of E_x for all $x \neq e$ is simply $||E||_1 - E_i$ and thus,

$$E[\tilde{E}_i] = E_i + \frac{||E||_1 - E_i}{M}$$

Taking an upper bound, we get:

$$E[\tilde{E}_i - E_i] \leq \frac{||E||_1}{M}$$

Notice that if $M = \frac{1}{\epsilon}$, we get our desired error.
Given this, our total space is $O\left(\frac{b}{\epsilon}\right)$

CountMin Sketch

The CountMin Sketch data structure is simply our solution but made more reliable via a process called **amplification** in which we make k *independent* (each with independent hash functions) copies of our data structure and run them in parallel. Then, when we query, we will take the minimum count of all k queries to reduce false positives. We can show that this will yield a good outcome *with high probability*.

The algorithm is simply a modification of our existing idea:

```
insert(i) {
        for all k:
                A_k[H_k(i)]++
}

delete(i) {
        for v = 1 to k:
                A_k[H_k(i)]--
}

lookup(e) {
        return min(A_k[H_k(i)] for all k)
}
```

Using Markov's Inequality, we can assert that for any k,

$$P\left(E[\tilde{E}_i - E_i] \geq \frac{2||E||_1}{M}\right) \leq \frac{1}{2}$$

Then, as the data structures are independent, the probability of the minimum being $\geq \frac{2||E||_1}{M}$ is

$$\leq \prod_{i=1}^{k} P(i^{th} \text{ structure has greater error})$$

$$\leq \frac{1}{2^k}$$

This means that the probability for success is $1 - \frac{1}{2^k}$

Setting $M = \frac{2}{\epsilon}$ yields our desired error once again.
Then, we can set a hyperparameter δ defined by the probability of a false positive.
If we set $k = lg\left(\frac{1}{\delta}\right)$, then the probability for success is exactly $1 - \delta$.

Then, for our high probability definition, we want the probability for failure to be $\frac{1}{n^c}$ for $c\log(n)$ tries.
Then, let's set $\delta = \frac{1}{n^c}$. This yields: $lg(n^c) = clg(n)$, which is what we wanted to show.

For space efficiency, we get $O(kMb) = O\left(lg\left(\frac{1}{\delta}\right)\frac{b}{\epsilon}\right)$

#streaming #algo #heavy-hitters #count-min-sketch #amplification