Streaming Algorithms

Today we'll talk about a topic that is both very old (as far as computer science goes), and very current. It's a model of computing where the amount of space we have is much less than the amount of data we examine, and hence we can store only a "summary" or "sketch" of the data. Back at the dawn of CS, data was stored on tapes and the amount of available RAM was very small, so people considered this model. And now, even when RAM is cheap and our machines have gigabytes of RAM and terabytes of disk space, it may be listening to an Ethernet cable carrying gigabytes of data *per second*, or training a machine learning model with terrabytes of training data. What can we compute in this model where our space is very limited compared to the size of the input?

Objectives of this lecture

In this lecture, we will:

- Introduce the data streaming model, and its concerns.
- Analyze an algorithm for heavy hitters in the arrivals-only model.
- Analyze an algorithm for heavy hitters with both arrivals and departures.

1 Introduction

Today's lecture will be about a slightly different computational model called the *data streaming* model. In this model you see elements going past in a "stream", and you have very little space to store things. For example, you might be running a program on an Internet router, the elements might be IP Addresses, and you have limited space. You certainly don't have space to store all the elements in the stream. Once you have read an element of the stream, you can not look back at prior elements of the stream. The question is: which functions of the input stream can you compute with what amount of time and space?

We will denote the stream elements by

$$a_1, a_2, a_3, \ldots, a_t, \ldots$$

We assume each stream element is from alphabet Σ and takes b bits to represent. For example, the elements might be 32-bit integers IP Addresses. We imagine we are given some function, and we want to compute it continually, on every prefix of the stream. Let us denote $a_{[1:t]} = \langle a_1, a_2, \ldots, a_t \rangle$.

Let us consider some examples. Suppose we have seen the integers

$$3,1,17,4,-9,32,101,3,-722,3,900,4,32,\ldots \qquad (\diamond)$$

Computing the sum Here, $F(a_{[1:t]}) = \sum_{i=1}^{t} a_i$. We want the outputs

$$3,4,21,25,16,48,149,152,-570,-567,333,337,369,\ldots$$

If we have seen T numbers so far, the sum is at most $T2^b$ and hence needs at most $O(b+\log T)$ bits of space. So we can keep a counter, and when a new element comes in, we add it to the counter.

Computing the max How about the maximum of the elements so far? $F(a_{[1:t]}) = \max_{i=1}^{t} a_i$. Even easier. The outputs are:

$$3,1,17,17,17,32,101,101,101,101,900,900,900$$

We just need to store b bits.

Computing the median The outputs on the various prefixes of $(\diamond)$ now are

$$3,1,3,3,3,3,4,3,\ldots$$

Doing this with small space is a lot more tricky.

The number of distinct elements Again, this is quite tricky to do exactly in low space.

Heavy hitters These are elements that have appeared most frequently in the stream.

You can imagine the applications of this model. An Internet router might see a lot of packets whiz by, and may want to figure out which data connections are using the most space? Or how many different connections have been initiated since midnight? Or the median (or the 90^{th} percentile) of the file sizes that have been transferred. Which IP connections are "elephants" (say the ones that have used more than 0.01% of your bandwidth)? Even if you are not working at "line speed",[1] but just looking over the server logs, you may not want to spend too much time to find out the answers, you may just want to read over the file in one quick pass and come up with an answer. Such an algorithm might also be cache-friendly. But how to do this? Two of the recurring themes will be:

- **Approximate solutions**: in several cases, it will be impossible to compute the function exactly using small space. Hence we'll explore the trade-offs between approximation and space. For example, we will develop algorithms that admit *false positives*, and algorithms that approximate the answers with some amount of error.
- **Hashing**: this will be a very powerful technique.

[1] Such a router might see tens of millions of packets per second.

***Remark: Measuring space in terms of* bits**

In this lecture, we will be focusing on space bounds rather than runtime. Additionally, in this model, we will be measuring the space usage of an algorithm in terms of the number of *bits* needed, rather than the number of words. In many other models, we would think of the space required to store n integers as $O(n)$ words. Outside the word RAM model, storing a sequence of n integers in the range 0 to n uses $O(n \log n)$ *bits* of space, because each integer requires $O(\log n)$ bits. In general, in this model, storing n integers of b bits each takes $O(nb)$ space. It is important to keep this in mind.

2 Finding ε-Heavy Hitters

Let's formalize things a bit. We have a data stream with elements $a_1, a_2, \ldots, a_t$ seen by time t. Think of these elements as "arriving" and being added to a multiset S_t, with $S_0 = \emptyset$, $S_1 = \{a_1\}$, $\ldots, S_i = \{a_1, a_2, \ldots, a_i\}$, etc. Let

$$\text{count}_t(e) = \{i \in \{1, 2, \ldots, t\} \mid a_i = e\}$$

be the number of times e has been seen in the stream so far. The *multiplicity* of e in S_t.

Definition 1: ε-heavy-hitters

Element $e \in \Sigma$ is called an *ε-heavy hitter* at time t if $\text{count}_t(e) > \varepsilon t$. That is, e constitutes *strictly more than* ε fraction of the elements that arrive by time t.

The goal is simple — given an threshold $\varepsilon \in [0, 1]$, maintain a data structure that is capable of outputting ε-heavy-hitters. At any point in time, we can query the data structure, and it outputs a set of at most $1/\varepsilon$ elements that contains all the ε-heavy hitters. At any moment in time, there are at most $1/\varepsilon$ elements that are ε-heavy-hitters, so this request is not unreasonable.

Remark

It's OK to output "false positives" but we are **not allowed** "false negatives", i.e., we're not allowed to miss any heavy-hitters, but we could output non-heavy-hitters. (Since we only output $1/\varepsilon$ elements, there can be $\leq 1/\varepsilon$-false positives.)

For example, if we're looking for $\frac{1}{3}$-heavy-hitters, and the stream is

$$E, D, B, D, D_{\ 5}, D, B, A, C, B_{\ 10}, B, E, E, E, E_{\ 15}, E \cdots$$

(the subscripts are not part of the stream, just to help you count) then

- at time 5, the element D is the only $\frac{1}{3}$-heavy-hitter,
- at time 11 both B and D are $\frac{1}{3}$-heavy-hitters, and
- at time 15, there is no $\frac{1}{3}$-heavy-hitter, and

- at time 16, only E is a $\frac{1}{3}$-heavy-hitter.

Note that as time passes, the set of frequent elements may change completely, so an algorithm would have to be adaptive. We cannot keep track of the counts for all the elements we've seen so far, there may be a lot of different elements that appear over time, and we have limited space.

Any ideas for how to find some "small" set containing all the ε-heavy-hitters?

Hmm, one trick that is useful in algorithm design, as in problem solving, is to try simple cases first. Can you find a 1-heavy-hitter? (Easy: there is no such element.) How about a 0.99-heavy-hitter? Random sampling will also work (maybe you'll see this in a homework or recitation), but let's focus on a deterministic algorithm for right now.

2.1 Finding a Majority Element

Let's first try to solve the problem of finding a 0.5-heavy-hitter, a.k.a. a majority element, an element that occurs (strictly) more than half the time. Keeping counts of all the elements is very wasteful! How can we find a majority element while using only a little space? Here's an algorithm (due to R. Boyer and J.S. Moore) that keeps very little information.

Algorithm 1: Boyer-Moore Majority

We keep one element from Σ, which is our candidate majority element, in a variable called `candidate`, and a counter

```
candidate = ⊥
counter = 0

When element a_t arrives
if (counter == 0)
  set candidate = a_t and counter = 1
else
  if a_t == candidate
    counter++
  else
    counter--
```

At the end, return `candidate`.

Intuitively, this algorithm works because a majority element occurs strictly more than half the time, and the counter variable is counting at least how many *more times* this element occurs than any other element, which must be a positive amount. It may, however, output a false positive when there does not exist a majority element.

- Suppose there is no majority element, then we'll output a false positive. As we said earlier, that's OK: we want to output a set of size 1 that contains the majority element, *if any*.
- If there is a majority element, we *will* indeed output it. Why? Observe: when we discard an element a_t, we also throw away another (different) element as well. (Decrementing the counter is like throwing away one copy of the element in `candidate`.) So every time we

throw away a copy of the majority element, we throw away another element too. Since there are fewer than half non-majority elements, we cannot throw away all the majority elements.

2.2 Finding an ε-heavy-hitter

We can extend this idea to finding ε-heavy-hitters.[2] Set $k = \lceil 1/\varepsilon \rceil - 1$; for $\varepsilon = 1/2$, we'd get $k = 1$.

Algorithm 2: ε-heavy-hitters

We keep an array Candidates$[1 \ldots k]$, where each location can hold one element from Σ; and an array Counts$[1 \ldots k]$, where each location can hold a non-negative integer.

```
Candidates[i]=⊥ for all 1≤i≤k
Counts[i]=0 for all 1≤i≤k

When element a_t arrives.
  if a_t==Candidates[j] for some j, then Counts[j]++.
  else if Counts[j]==0 for some j then Candidates[j]←a_t and Counts[j]←1.
  else decrement all the counters by 1.
```

An an exercise, check that this algorithm is identical to the one above for $\varepsilon = 1/2$.

To analyze the algorithm and prove that it is correct, we define the *estimated count* which corresponds to the current count that the algorithm is maintaining for each of the candidates (or zero for those elements which are not currently a candidate):

$$\text{est}_t(e) = \begin{cases} \text{Counts}[j] & \text{if } e == \text{Candidates}[j] \\ 0 & \text{otherwise} \end{cases}$$

The main claim naturally extends the proof for the case of majority.

Lemma 1

The estimated counts satisfy:

$$0 \le \text{count}_t(e) - \text{est}_t(e) \le \frac{t}{k+1} \le \varepsilon t$$

Proof. The estimated count $\text{est}_t(e)$ is at most the actual count, since we never increase a counter for e unless we see e. So $\text{count}_t(e) - \text{est}_t(e) \ge 0$. (In other words, $\text{est}_t(e) \le \text{count}_t(e)$, it is always an underestimate.)

To prove the other inequality, think of the arrays Counts and Candidates saying that we have Counts[1] copies of Candidates[1], Counts[2] copies of Candidates[2], etc., in our hand. Look at some time when the difference between $\text{count}_t(e)$ and $\text{est}_t(e)$ increases by 1. Each time

[2] The extension to finding ε-heavy-hitters was given by J. Misra and D. Gries, and independently by R.M. Karp, C.H. Papadimitriou and S. Shenker.

this happens, we decrement k different counters by 1 and discard an element (which is not currently present in the candidates array). This is like dropping $k+1$ distinct elements (one of which is e). We can drop at most t elements until time t. So the gap can be at most $t/(k+1)$. And since $k = \lceil 1/\varepsilon \rceil - 1 \geq 1/\varepsilon - 1$, we get that $t/(k+1) \leq \varepsilon t$. □

Corollary 1

For every n, the set of items in the array T contains all the ε-heavy-hitters.

Proof. After we've seen n elements, the ε-heavy-hitters have occurred at least $\text{count}_t(e) > \varepsilon t$ times. If e is a ε-heavy-hitter, by Lemma 1, the estimate

$$\text{est}_t(e) \geq \text{count}_t(e) - \varepsilon t > 0.$$

So element e must be in the array T. □

To summarize: if we set $k = \lceil 1/\varepsilon \rceil - 1$, we get an algorithm that gives us element counts on a stream of length t to within an additive error of at most $(\varepsilon \cdot t)$ with space $O(1/\varepsilon) \cdot (\log \Sigma + \log t)$ bits. As a corollary we get an algorithm to find a set containing the ε-heavy-hitters.

3 Heavy Hitters with Deletions

In the above problem, we assumed that the elements were only being added to the current set at each time. We maintained an approximate count for the elements (up to an error of εt). Now suppose we have a stream where elements can be both added and removed from the current set. How can we give estimates for the counts?

Formally, each time we get an *update*, it looks like (add, e) or (del, e). We will assume that for each element, the number of deletes we see for it is at most the number of adds we see — the running counts of each element is non-negative. As an example, suppose the stream looked like:

$$(\text{add}, A), (\text{add}, B), (\text{add}, A), (\text{del}, B), (\text{del}, A), (\text{add}, C)$$

then the sets at various times are

$$S_0 = \emptyset, S_1 = \{A\}, S_2 = \{A, B\}, S_3 = \{A, A, B\}, S_4 = \{A, A\}, S_5 = \{A\}, S_6 = \{A, C\}, \ldots$$

The counts of the element are defined in the natural way: $\text{count}_3(A) = 2$ and $\text{count}_5(A) = 1$. Observe that the "active" set S_t has size $|S_t| = \sum_{e \in \Sigma} \text{count}_t(e)$, and its size can grow and shrink.

What do we want want now? We want a data structure to answer count queries approximately. Specifically, at any point in time t, for any element, we should be able to ask "What is $\text{count}_t(e)$?" The data structure should respond with an estimate $\text{est}_t(e)$ such that

$$\Pr\Big[\underbrace{|\text{est}_t(e) - \text{count}_t(e)|}_{\text{error}} \leq \underbrace{\varepsilon |S_t|}_{\text{is small}}\Big] \geq \underbrace{1-\delta}_{\text{with high probability}}.$$

Again, we would again like to use small space — perhaps even close to the

$$O(1/\varepsilon)\cdot(\log\Sigma+\log|S_t|) \text{ bits of space}$$

we used in the above algorithm. (As you'll see, we'll get close.)

3.1 A Hashing-Based Solution: First Cut

We're going to be using hashing for this approach, simple and effective. We'll worry about what properties we need for our hash functions later, for now assume we have a hash function $h : \Sigma \to \{0,1,\ldots,k-1\}$ for some suitably large integer k.

Algorithm 3: Approximate count

Maintain an array counts$[1\ldots k]$ capable of storing non-negative integers.

```
when update a_t arrives
  if (a_t == (add, e)) then
    counts[h(e)]++;
  else // a_t == (del, e)
    counts[h(e)]--;
```

One way to think about this is that we are just maintaining a hashtable *without collision resolution*. This was the update procedure. And what is our estimate for the number of copies of element e in our active set S_t? It is

$$\text{est}_t(e) := \text{counts}[h(e)].$$

In words, we look at the location $h(e)$ where e gets mapped using the hash function h, and look at counts$[h(x)]$ stored at that location. What does it contain? It contains the current count for element e for sure. But added to it is the current count for any other element that also gets mapped to that same location. In math:

$$\text{counts}[h(e)] = \sum_{e'\in\Sigma} \text{count}_t(e')\cdot\mathbf{1}(h(e')=h(e)),$$

where $\mathbf{1}$(some condition) is a function that evaluates to 1 when the condition in the parentheses is true, and 0 if it is false. We can rewrite this as

$$A(h(e)) = \text{count}_t(e) + \sum_{e'\neq e} \text{count}_t(e')\cdot\mathbf{1}(h(e')=h(e)),$$

or using the definition of the estimate, as

$$\text{est}_t(e) - \text{count}_t(e) = \sum_{e'\neq e} \text{count}_t(e')\cdot\mathbf{1}(h(e')=h(e)). \tag{1}$$

A great situation will be if no other elements $e'\neq e$ hashed to location $h(e)$ and the error (i.e., the summation on the right) evaluates to zero. But that may be unlikely. On the other hand, we can show that the expected error is not too much.

What's the expected error? Now we need to assume something good about the hash functions. Assume that the hash function h is a random draw from a universal family. Recall the definition from earlier in the course:

Definition: Universal hashing

A family $\mathcal{H}$ of hash functions from $\Sigma \to [k]$ is universal if for any pair of keys $x_1, x_2 \in \Sigma$ with $x_1 \neq x_2$,

$$\Pr[h(x_1) = h(x_2)] \leq \frac{1}{k}.$$

In other words, if we just look at two keys, the probability that they collide is no more than if we chose to map them randomly into the range $[k]$. We gave a construction where each hash function in the family used $(\lg k) \cdot (\lg |\Sigma|)$ bits to specify.

Good. So we drew a hash function h from this universal hash family $\mathcal{H}$, and we used it to map elements to locations $\{0, 1, \ldots, k-1\}$. What is its expected error? From (1), it is

$$\begin{aligned} E\left[\sum_{e' \neq e} \text{count}_t(e') \cdot \mathbf{1}(h(e') = h(e))\right] &= \sum_{e' \neq e} \text{count}_t(e') \cdot E\left[\mathbf{1}(h(e') = h(e))\right] \quad (2) \\ &= \sum_{e' \neq e} \text{count}_t(e') \cdot \Pr[h(e') = h(e)] \\ &\leq \sum_{e' \neq e} \text{count}_t(e') \cdot (1/k) \quad (3) \\ &= \frac{|S_t| - \text{count}_t(e)}{k} \leq \frac{|S_t|}{k}. \end{aligned}$$

We used linearity of expectations in equality (2). To get (3) from the previous line, we used the definition of a universal hash family. Let's summarize:

Claim 1

The estimator $\text{est}_t(e) = \text{counts}[h(e)]$ ensures that

(a) $\text{est}_t(e) \geq \text{count}_t(e)$, and

(b) $E[\text{est}_t(e)] - \text{count}_t(e) \leq |S_t|/k$.

The space used is: k counters, and $O((\log k)(\log \Sigma))$ to store the hash function.

That's pretty awesome. (But perhaps not so surprising, once you think about it.) Note that if we were only doing arrivals and no deletions, the size of S_t would be exactly t. So the expected error would be at most t/k, which is about the same as we were getting in Section 2 using an array of size $(k-1)$. But now we can also handle deletions!

What's the disadvantage? We only have a bound on the *expected error* $E[\text{est}_t(e)] - \text{count}_t(e)$. It is no longer deterministic. Also, the expected error being small is weaker than saying: we have small error with high probability. So let's see how to improve things.

3.2 Amplification of the Success Probability

Any ideas how to *amplify* the probability that we are close to the expectation? The idea is simple: *independent repetitions*. Let us pick m hash functions $h_1, h_2, \ldots, h_m$. Each $h_i : \Sigma \to \{0, 1, \ldots, k-1\}$. How do we choose these hash functions? *Independently* from the universal hash family H.[3]

Algorithm: CountMin sketch

We have m arrays $\text{counts}_1, \text{counts}_2, \ldots, \text{counts}_m$, one for each hash function. The algorithm now just uses the i^{th} hash function to choose a location in the i^{th} array, and increments or decrements the same as before.

```
when update a_t arrives
  for each i from 1..m
    if (a_t == (add, e)) then
      counts_i[h_i(e)]++
    else // a_t == (delete, e)
      counts_i[h_i(e)]--
```

And what is our new estimate for the number of copies of element e in our active set? It is

$$\text{best}_t(e) := \min_{i=1}^{m} \text{counts}_i[h_i(e)].$$

In other words, each (h_i, counts_i) pair gives us an estimate, and we take the least of these. It makes perfect sense — the estimates here are all *over*estimates, so taking the least of these is the right thing to do. But how much better is this estimator? Let's do the math.

What is the chance that one single estimator has error more than $2|S_t|/k$? Remember the expected error is at most $|S_t|/k$. So by Markov's inequality

$$\Pr\left[\text{error} > 2 \cdot \frac{|S_t|}{k}\right] \leq \frac{1}{2}.$$

And what is the chance that all of the m copies have "large" (i.e., more than $2|S_t|/k$) error? The probability of m failures is

$$\begin{aligned}
&\Pr[\text{each of } m \text{ copies have large error}] \\
&= \prod_{i=1}^{m} \Pr[i^{th} \text{ copy had large error}] \\
&\leq (1/2)^m .
\end{aligned}$$

The first equality there used the independence of the hash function choices. (Only if events $\mathscr{A}, \mathscr{B}$ are independent you can use $\Pr[\mathscr{A} \wedge \mathscr{B}] = \Pr[\mathscr{A}] \cdot \Pr[\mathscr{B}]$.) And so the minimum of the estimates will have "small" error (i.e., at most $2|S_t|/k$) with probability at least $1-(1/2)^m$.

[3] If we use the random binary matrix hash family construction given in the hashing lecture, this means the $(\lg k) \cdot (\lg |\Sigma|)$-bit matrices for each hash function must be filled with independent random bits.

3.2.1 Final Bookkeeping

Let's set the parameters now. Set $k = 2/\varepsilon$, so that the error bound $2|S_t|/k = \varepsilon|S_t|$. And suppose we set $m = \lg 1/\delta$, then the failure probability is $(1/2)^m = \delta$, and our query will succeed with probability at least $1-\delta$. [4]

Then on any particular estimate $\text{best}_t(e)$ we ensure

$$\Pr\left[\left|\text{best}_t(e) - \text{count}_t(e)\right| \leq \varepsilon|S_t|\right] \geq 1-\delta.$$

Just as we wanted. And the total space usage is

$$m \cdot k \text{ counters} = O(\log 1/\delta) \cdot O(1/\varepsilon) = O(1/\varepsilon \log 1/\delta) \text{ counters.}$$

Each counter has to store at most $\lg T$-bit numbers after T time steps. [5]

Space for Hash Functions: We need to store the m hash functions as well. How much space does that use? The random binary matrix method that we learned earlier in the course used $s := (\lg k) \cdot (\lg \Sigma)$ bits per hash function. Since $k = 1/\varepsilon$, the total space used for the functions is

$$m \cdot s = O(\log 1/\delta) \cdot (\lg 1/\varepsilon) \cdot (\lg \Sigma) \text{ bits.}$$

3.2.2 And in Summary...

Using about $1/\varepsilon \times$ poly-logarithmic factors space, and very simple hashing ideas, we could maintain the counts of elements in a data stream under both arrivals and departures (up to an error of $\varepsilon|S_t|$). As in the arrival-only case, these counts make sense only for very high-frequency elements.

[4] How small should you make δ? Depends on how many queries you want to do. Suppose you want to make a query a million times a day, then you could make $\delta = 1/10^9 \approx 1/2^{30}$ a 1-in-1000 chance that even one of your answers has high error. Our space varies linearly as $\lg 1/\delta$, so setting $\delta = 1/10^{18}$ instead of $1/10^9$ doubles the space usage, but drops the error probability by a factor of billion.

[5] So a 32-counter can handle a data stream of length 4 billion. If that is not enough, there are further techniques to reduce this space usage as well.