CS 261: Graduate Data Structures

Week 4: Streaming and sketching

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Streaming and sketching

The main idea

Sometimes data is too big to fit into memory...

Sketching

Represent data by a structure of sublinear size

Preferably $O(1)$; at most $n^{1-\varepsilon}$ for some $\varepsilon > 0$

Keep enough information to solve what you want either exactly or approximately

Streaming

Read through your data once, in some given order

Use a sketch to represent information about it

Produce a result from the sketch

Example: Sketch for the average

Sketch for the average of a collection of numbers: pair (n, t)

n is the number of elements in the collection
t is the sum of the elements

Operations:

- Incorporate new element x into collection:
  Add one to n and add x to t
- Remove element from collection:
  Subtract one from n and subtract x from t
- Compute average of collection:
  Return t/n

Total space is $O(1)$ regardless of how big n is

(Time/operation is also $O(1)$ but our main concern is space)

Example: Streaming average

To compute the average of a sequence S:

- Initialize an empty sketch for the average
- For each x in the sequence, incorporate x into the sketch
- Return the average of the the sketch

Time for an n-element sequence is $O(n)$
Working space (not counting the space for the input) is $O(1)$

This works even when the sequence is generated somehow in a way that does not involve local storage (e.g. internet traffic)

Cash register versus turnstile

Two different variations of streaming algorithms:

Cash register: Money goes in but doesn't come back out

Meaning for streaming:

Input sequence has only insertions, no removals

Turnstile: Counts people passing in either direction

Meaning for streaming:

Input sequence can have both insertions and removals

Streaming average was presented in cash register model but the same sketch also works for turnstile model

Images: File:Credit card terminal in Laos.jpg and File:Warszawa - Metro - witokrzyska (17009062241).jpg from Wikimedia commons

Beyond streaming

Sketches can sometimes be combined to produce information about multiple data sets

Example: Can produce an average-sketch for the concatenation of two sequences by adding the sketches of the two sequences

If we have n sequences, of total length N, we can compute all pairwise averages in time $O(n^2 + N)$, faster than computing each pairwise average separately

Medians and reservoir sampling

Medians

Median: Sort given list of n items; which one goes to position $n/2$?

Like average, an important measure of the central tendency of a sample, more robust against arbitrarily-corrupted outliers

Applications in facility location: Given n points on a line, place a center minimizing the average distance to the given points

Impossibility of streaming median

No streaming algorithm can compute the median!

Consider sketch after seeing the first $n/2$ items in a stream

Depending on the remaining items, any one of these first $n/2$ could become the median of the whole stream

So we have to remember all their values

Space is $\Omega(n)$, not $O(n^{1-\varepsilon})$ for some $\varepsilon > 0$

Approximate medians

We want the item at position $\frac{1}{2}n$ in sorted sequence

Relaxation: δ-approximate median

Position should be between $(\frac{1}{2} - \delta)n$ and $(\frac{1}{2} + \delta)n$

Easy randomized method: Choose a sample of $\Theta(\delta^{-2})$ sequence members and return median of sample

Chernoff $\Rightarrow$ constant probability of being in desired position

Bigger constant in sample size $\Rightarrow$ better probability

(There also exist non-random solutions but more complicated and with non-constant space complexity)

Reservoir sampling

How to maintain a sample of k random elements (without replacement) from a longer sequence?

Store an array A of size k, and the number n
For each element x:
    If $n < k$:
        Store x in $A[n]$
    Else:
        Choose a random integer i from 0 to $n - 1$
        If $i < k$, store x in $A[i]$
    Increment n

(More complex methods reduce the number of random number generation steps by calculating, after each replacement, when the next replacement should be made.)

Majority and heavy hitters

Impossibility of most-frequent item

We've seen the mean and median; what about the mode (most frequent item) in a sequence?

Impossible in general!

Consider the state of a sketch after $n-2$ items

Suppose we have already seen one item appear three times, and $(n-3)/2$ pairs of other items

If the next two items are equal, with value x, then the result should be

- The triple, if x was not already seen
- The triple, if x has the same value
- x, if x has the same value as one of the pairs

So the sketch must know all the pairs, too much memory

Impossibility of majority

Majority element: one that occurs as more than half of the sequence elements

Naive formulation of the problem:

Either find a majority element if one exists

Or return "None" if there is no majority

Still impossible!

If first $n/2$ elements are distinct, any one could become majority

We don't have enough memory to store all of their values

Boyer–Moore majority vote algorithm

Maintain two variables, m and c, initially None and 0

For each x in the given sequence:

- If $m = x$: increment c
- Else if $c = 0$: set $m = x$ and $c = 1$
- Else decrement c

Claims:

- If sequence has a majority, it will be the final value of m
- If no majority, m may be any element of the sequence (might not be frequently occurring)
- Algorithm cannot tell us whether m is majority element or not

Boyer & Moore, "MJRTY - A Fast Majority Vote Algorithm", 1991

Streaming majority example

With the sequence elements in left-to-right order:

x:		B	A	C	A	A	A	C	B	A
m:	None	B	B	C	C	A	A	A	A	A
c:	0	1	0	1	0	1	2	1	0	1
majority?	N	Y	N	N	N	Y	Y	Y	N	Y

(We know when there is a majority but the algorithm doesn't)

After the first step, B is in the majority, and the algorithm correctly reports B as its value of m.

After the third A (out of five sequence elements), A is in the majority, and stays in the majority for most of the remaining steps; the algorithm correctly reports A as its value of m.

When there is no majority (each element has at most half of the sequence) the algorithm may choose any element as its value of m.

Heavy hitters

Generalization of the same algorithm:

Store two arrays M and C, both of length k

Initialize cells of M to empty and C to zero

For each sequence item x:

- If x equals $M[i]$ for some i: increment $C[i]$
- Else if some $C[i]$ is zero: set $M[i] = x$ and $C[i] = 1$
- Else decrement $C[i]$ for all choices of i

Claim: In a sequence of n elements, if x occurs more than $n/(k+1)$ times, then x will be one of the elements stored in M

(Special case $k = 1$ is correctness of majority algorithm)

Why does it work?

This algorithm approximately counts occurrences for all sequence elements (not just the ones it stores)!

Define count(x) = actual number of occurrences of x,
estimate(x) = $C[i]$ (if $M[i] = x$ for some i) or 0 otherwise.

Then $\text{count}(x) - \frac{n}{k+1} \leq \text{estimate}(x) \leq \text{count}(x)$.

Proof idea: Induction on number of steps of the algorithm

If a step ends with x in M, we increase its count and estimate

Otherwise, we decrease all $k+1$ nonzero estimates

Number of increases $\leq n \Rightarrow$ number of decreases $\leq n/(k+1)$

Because heavy hitters have big estimates, they must be stored

MinHash

Hashing for reservoir sampling

We saw how to maintain a sample of k elements, using a random choice for each element

Instead, make a single random choice, of a hash function $h(x)$

Sample = the k elements with the smallest values of h

Two advantages:

- Less use of (possibly slow) random number generation
- Repeatable: if different agents use the same hash function to sample the same sets, they will get the same samples

We can take advantage of repeatability to use this for more applications!

Broder, "On the resemblance and containment of documents", 1997

Combining sets with MinHash

To compute the union:

$$\text{MinHash}(S \cup T) = \text{MinHash}\big(\text{MinHash}(S) \cup \text{MinHash}(T)\big)$$

Because any element that is one of the k smallest hashes in the union must also be one of the k smallest in either of the smaller sets that contain it

We can get useful information about sets S and T from this computation!

Key question: how many elements from $\text{MinHash}(S) \cup \text{MinHash}(T)$ survive into $\text{MinHash}(S \cup T)$?

Original motivation for MinHash

Before the success of Google, a different company (DEC) ran a different popular search engine (AltaVista)

They had a problem: Many web pages can be found in multiple similar (but not always identical) copies

Search engines should return only one copy of the same page, so user can see other results without getting swamped by duplicates

$\Rightarrow$ Need to quickly test similarity of web pages

From documents to sets

"Bag of words" model: represent any web page by the set of words used in it

Similar documents (web pages) should have similar words

So after converting web pages to bags of words, the search engine problem becomes: quickly test similarity of pairs of sets

Jaccard similarity

A standard measure of how similar two sets are:

$$J(S, T) = \frac{|S \cap T|}{|S \cup T|}$$

Scale-invariant (normalized for how big the two sets are)

Ranges from 0 to 1

- 0 when sets are disjoint
- close to 0 when sets are not very similar
- close to 1 when sets are similar
- 1 when sets are identical

Estimating Jaccard similarity

Theorem:

$$J(S,T) = \frac{1}{k} E\Big[\big|\text{MinHash}(S \cup T) \cap \text{MinHash}(S) \cap \text{MinHash}(T)\big|\Big]$$

The intersection consists of the elements of $\text{MinHash}(S \cup T)$ that also belong to $\text{MinHash}(S \cap T)$

Linearity of expectation $\Rightarrow$

$$\begin{aligned}
E[\ldots] &= \sum_{x \in S \cup T} \Pr[x \in \text{intersection of MinHashes}] \\
&= \sum_{x \in S \cup T} \Pr[x \in S \cap T] \times \Pr[x \in \text{MinHash}(S \cup T)] \\
&= \sum_{x \in S \cup T} \frac{|S \cap T|}{|S \cup T|} \times \frac{k}{|S \cup T|} \\
&= |S \cup T| \times J(S,T) \times \frac{k}{|S \cup T|} = k\, J(S,T)
\end{aligned}$$

How accurate is it?

Suppose we use

$$\frac{1}{k}\left|\text{MinHash}(S \cup T) \cap \text{MinHash}(S) \cap \text{MinHash}(T)\right|$$

as an estimate for $J(S, T)$

Previous analysis shows that this has the correct expected value: it is an *unbiased estimator*

Our analysis decomposed the estimate into a sum of independent 0-1 random variables (is each member of $S \cup T$ in the intersection)

$\Rightarrow$ Can use Chernoff bounds for being within $1 \pm \varepsilon$ of expectation

To get expected error $|J(S, T) - \text{estimate}| \leq \varepsilon$, set $k = \Theta(1/\varepsilon^2)$

Summary of MinHash

Main idea: sketch any set by choosing the k elements that have the smallest hash values

Can reduce randomness in reservoir sampling

Sketches of unions can be computed from unions of sketches

Provides accurate estimates of Jaccard similarity

All-pairs similarity of n sets with total length N: time $O(n^2 + N)$

Count-min sketch

Main idea

We want to estimate frequencies of individual items (like heavy hitter data structure) in the turnstile model, allowing removals

Use hashing to map each item to a multiple cells in a small table
(like Bloom filter, but smaller)

Each cell counts how many elements map to it

- Include another copy of x: increment all cells for x
- Remove a copy of x: decrement all cells for x

Estimate of frequency of x: minimum of counts in cells for x

Cormode & Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications", 2005

Details

Set up according to two parameters ε and δ

- ε: how accurate the estimates should be
- δ: how likely estimates are to be accurate

2d table $C[i, j]$, initially all zero

- Horizontal dimension (i): $\lceil e/\varepsilon \rceil$ cells
- Vertical dimension (j): $\lceil \ln 1/\delta \rceil$ cells

Hash functions h_j (for $j = 0, \ldots \lceil \ln 1/\delta \rceil - 1$)

- Include x: increment $C[h_j(x), j]$ for all j
- Remove x: decrement $C[h_j(x), j]$ for all j

Estimate of count(x): $\min_j C[h_j(x), j]$

Accuracy

Clearly, estimate $\geq$ count; let $N =$ total number of elements

Claim: With probability $\geq 1 - \delta$, estimate $\leq$ count $+ \varepsilon N$

Because row estimates are independent, this is equivalent to:

With probability $\geq 1 - \frac{1}{e}$, for any j, $C[h_j(x), j] \leq$ count $+ \varepsilon N$

(then multiplying row failure probabilities, $(1/e)^{\text{number of rows}} \leq \delta$)

Overestimate comes from collisions with other elements

Expected number of collisions $= \epsilon N/e$. Instead of Chernoff, use Markov's inequality: $Pr[X > c\,E[X]] \leq 1/c$

Turnstile medians

Data structure for approximate medians in the turnstile model, of a collection S of integers $0, 1, 2, \ldots n$ (allowing repetitions):

Form sets S_i ($i = 0 \ldots \log_2 n$) by rounding each element down to a multiple of 2^i, using the formula `round(x) = x &~ ((1<<i)-1)`

Construct $\log n$ count-min sketches, one for each S_i

(with accuracy and failure probability smaller by a logarithmic factor so that when we query all of them and sum, the overall failure probability and accuracy are still what we want)

Finding the approximate median

To estimate the number of elements in the interval $[\ell, r]$:

Compute the numbers

- $\ell_i = \text{round}(\ell - 1) + 1$
- $r_i = \text{round}(r)$

The query interval can be decomposed into subintervals of length 2^i, consisting of everything that rounds to ℓ_i or r_i, for a subset of these numbers

Estimate occurrences of ℓ_i and r_i in S_i and sum the results

To find median: binary search for an interval $[0, m]$ containing approximately half of the elements

Set reconciliation

Find the missing element

Toy problem:

I give you a sequence of 999 different numbers, from 1 to 1000

Which number is missing?

Streaming solution: return $500500 - \sum(\text{sequence})$

Straggler sketching

Given a parameter k, maintain a turnstile collection of objects (allowing repetitions), such that

- Total space is $O(k)$
- The set can have arbitrarily many elements
- Whenever it has $\leq k$ distinct elements, we can list them all (with their numbers of repetitions)

E.g. could solve the toy problem by:

- Initialize structure with $k = 1$
- Insert all numbers from 1 to 1000
- Remove all members of sequence
- Ask which element is left

Invertible Bloom filter

Structure:

- Table with $O(k)$ cells
- $O(1)$ hash functions $h_i(x)$ mapping elements to cells (like a Bloom filter)
- "Fingerprint" hash function $f(x)$ (like a cuckoo filter)

Each cell stores three values:

- Number of elements mapped to it (like count-min sketch)
- Sum of elements mapped to it
- Sum of fingerprints of elements mapped to it

Eppstein & Goodrich, "Space-efficient straggler identification in round-trip data streams via Newton's identitities and invertible Bloom filters", 2011

How to list the elements

"Pure cell": stores only a single distinct element (but maybe more than one copy of it)

If a pure cell stores element x, we can compute x as
(sum of elements)/(number of elements)

Test whether a cell is pure: compute x and check that stored sum of fingerprints equals number of copies times fingerprint(x)

Decoding algorithm:

- While there is a pure cell, decode it, output it, and remove its elements from the collection
- If we reach an empty table (all cells zero), return
- Otherwise, decoding fails (table is too full)

Application: Set reconciliation

Problem: Two internet servers both have slightly different copies of the same collection (e.g. versions of a git repository)

We want to find how they differ, using communication proportional to the size of the difference (rather than the size of the whole set)

If we already know the size of the difference, k:

- Server A builds an invertible Bloom filter for its collection, with parameter k, and sends it to server B
- Server B removes everything in its collection from the filter and then decodes the result
- (Some elements might have negative numbers of copies! But the data structure still works)

Eppstein, Goodrich, Uyeda, and Varghese, "What's the difference? Efficient set reconciliation without prior context", 2011

How to estimate the size of the difference?

Use MinHash to find samples of server B's set with numbers of elements smaller by factors of $1/2$, $1/4$, $1/8$, ...

Sample i: elements whose hash has first i bits = zero

Server B chooses parameter $k = O(1)$ and sends invertible Bloom filters of all the samples in a single message to server A

Server A uses the same hash function to sample its set, and removes sampled subsets from B's filters

Estimate of the size of the difference = $1/$sampling rate of largest sample whose difference can be decoded

Summary

Summary

- Sketch: Structure with useful information in sublinear space
- Stream: Loop through data once, using a sketch
- Cash register (addition only) and turnstile (addition and removal) models of sketching and streaming
- Mean and median; impossibility of exact median
- Maintaining a random sample of a stream and using it for approximate medians
- Majority voting, heavy hitters, and frequency estimation
- MinHash-based sampling and estimation of Jaccard distance
- Count-min sketch turnstile-model frequency estimation
- Using count-min sketch for turnstile median estimation
- Stragglers, invertible Bloom filters, and set reconciliation