Lecture 2. Frequency problems

Ricard Gavaldà

MIRI Seminar on Data Streams, Spring 2015

Contents

Frequency problems in data streams

The data stream model. Frequency problems

- Input is sequence of items $a_1, a_2, a_3, \ldots$
- Each a_i is an element of a universe U of size n
- n is large or infinity

- At time t, the query returns something about $a_1 \ldots a_t$

Frequency problems, alternate view

- At any time t, for any $i \in U$,

  $$f_i = \text{(def.) number of appearances of } i \text{ so far}$$

- Frequency problems: result depends on the f_i's only
- In particular, independent of the order

Frequency problems, alternate view

- Stream at t defines implicit array $F[1..n]$ with $F[i] = f_i$
- A new occurrence of i equivalent to "$F[i]$++"
- Model extensions:
  - $F[i]$++, $F[i]$-- (additions and deletions)
  - $F[i]$ += x, with $x \geq 0$ (multiple additions)
  - $F[i]$ += x, with any x (multiple additions and deletions)

Approximating inner product

Approximating inner product

- Implicit vectors $u[1..n]$, $v[1..n]$
- Stream of instructions "add(u_i,x)", "add(v_j,y)", $i, j = 1 \dots n$
- At every time, we want to output an approximation of

$$\sum_{i=1}^{n} u_i \cdot v_i$$

- I'll suppose the above is always > 0 for relative approximation to make sense

Basic algorithm

Init:

- Pick a very "good" hash function $f : [n] \to [n]$
- For $i \in [n]$, define (do *not* compute and store)

$$b_i = (-1)^{f(i) \bmod 2} \in \{1, -1\}$$

- $S \leftarrow 0$; $T \leftarrow 0$;

Update:

- When reading "add(u_i,x)", do S += $x \cdot b_i$
- When reading 'add(v_j,y)", do T += $y \cdot b_j$

Query:

- return $S \cdot T$

Final algorithm

- Run in parallel $c_1 \cdot c_2$ copies of the basic algorithm, grouped in c_2 groups of c_1 each
- When queried, compute the average of the results of each group of c_1 copies, then return the median of the averages of the c_2 groups

Theorem

For $c_1 = O(\varepsilon^{-2})$ and $c_2 = O(\ln \delta^{-1})$, the algorithm above (ε, δ)-approximates $u \cdot v$

Why does this work?

- Claim 1: $S = \sum_{i=1}^{n} u_i b_i$ and $T = \sum_{i=1}^{n} v_i b_i$
- Claim 2: $E[S \cdot T] = IP(u, v)$
- Claim 3: $Var[S \cdot T] \leq 2\,E[S \cdot T]^2$
- Claim 4: The median-of-averages as described (ε, δ)-approximates $IP(u, v)$

Claim 1

Claim 1: $S = \sum_{i=1}^{n} u[i]b_i$ and $T = \sum_{i=1}^{n} v[i]b_i$

Update is:

- When reading “add(u_i,x)”, do S += $x \cdot b_i$
- When reading ‘add(v_j,y)”, do T += $y \cdot b_j$

Claim 2

Claim 2: $E[S \cdot T] = IP(u, v)$

Really? But

$$S = \left(\sum_i u_i b_i\right), \quad T = \left(\sum_i v_i b_i\right)$$

yet

$$\left(\sum_i u_i\right) \cdot \left(\sum_i v_i\right) = \left(\sum_{i,j} u_i v_j\right) \neq \left(\sum_i u_i v_i\right)$$

So the trick has to be in the b_i, b_j

Claim 2 (II)

- If $i = j$, $E[b_i b_j] = E[1] = 1$
- If $i \neq j$ and h is "good", b_i and b_j are independent, so

$$E[b_i b_j] = \frac{1}{2}1 + \frac{1}{2}(-1) = 0$$

Then Claim 2 is by linearity of expectation:

$$\begin{aligned}
E[S \cdot T] &= E[\left(\sum_{i=1}^{n} u[i] b_i\right)\left(\sum_{i=1}^{n} v[i] b_i)\right)] \\
&= E[\sum_{i,j} u[i] v[j]\, b_i b_j\,] \\
&= \sum_{i} u[i] v[i]\, E[b_i b_i] + \sum_{i \neq j} u[i] v[j]\, E[b_i b_j] \\
&= \sum_{i} u[i] v[i]
\end{aligned}$$

Claim 3

Claim 3: $Var[S \cdot T] \leq 2\,E[S \cdot T]^2$

$$
\begin{aligned}
Var[S \cdot T] &= E[(S \cdot T)^2] - E[S \cdot T]^2 \\
&= (\sum_{i,j} \ldots b_i b_j \ldots) \cdot (\sum_{k,\ell} \ldots b_k b_\ell \ldots) \\
&= \sum_{i,j,k,\ell} (\ldots b_i b_j b_k b_\ell \ldots) \\
&\leq 2(\sum_i u[i]v[i]) \cdot (\sum_j u[j]v[j]) \\
&= 2\,E[S \cdot T]^2
\end{aligned}
$$

(you work it out)

Claim 4

Claim 4: Average c_1 copies of $S \cdot T$

- Let X be the output of the basic algorithm
  - $E[X] = IP(u, v)$, $Var(X) \leq 2E[X]^2$
  - Equivalently, $\sigma(X) = \sqrt{Var(X)} \leq \sqrt{2}E[X]$
- Want to bound $\Pr[|X - E[X]| > \varepsilon E[X]]$

$$\Pr[|X - E[X]| > \varepsilon E[X]] \leq \Pr[|X - E[X]| > \sqrt{2}\varepsilon\sigma(X)]$$

But applying Chebyshev requires $\sqrt{2}\varepsilon > 1$, not interesting
We need to reduce the variance first: averaging

Claim 4 (cont.)

- Let X_i be the output of i-th copy of basic algorithm
  - $E[X_i] = IP(u,v)$, $Var(X_i) \leq 2E[X_i]^2$
- Let Y be the average of $X_1, \ldots, X_{c_1}$
- See that $E[Y] = IP(u,v)$ and $Var(Y) \leq 2E[Y]^2/c_1$
- By Chebyshev's inequality, if $c_1 \geq 16/\varepsilon^2$

$$\begin{aligned} \Pr[|Y - E[Y]| > \varepsilon E[Y]] &\leq Var(Y)/(\varepsilon E[Y])^2 \\ &\leq 2E[Y]^2/(c_1 \varepsilon^2 E[Y]^2) \leq 1/8 \end{aligned}$$

We could throw δ into this bound, but get dependence $1/\delta$
At this point, use Hoeffding to get $\ln(1/\delta)$

Claim 4 (cont.)

We have $E[Y] = IP(u,v)$ and

$$\Pr[(1-\varepsilon)E[Y] \leq Y \leq (1+\varepsilon)E[Y]] \geq 7/8$$

Now take the median Z of c_2 copies of Y, $Y_1, \ldots, Y_{c_2}$
As in the exercise on computing medians (Hoeffding bound),

$$\Pr[|Z - E[Y]| \geq \varepsilon E[Y]] \leq \delta$$

if

$$c_2 \geq \frac{32}{9} \ln \frac{2}{\delta}$$

We get (ε, δ)-approximation with

$$c_1 \cdot c_2 = O\left(\frac{1}{\varepsilon^2} \ln \frac{2}{\delta}\right)$$

copies of the basic algorithm

Memory use & update time

- $c = O(\frac{1}{\varepsilon^2} \ln \frac{1}{\delta})$ copies of algorithm
- Each, $4 \log n$ bits to store hash function
- At most $\log \sum_i u_i + \log \sum_i v_i$ bits to store S, T
- Say, $O(\log t)$ if the u_i, v_i are bounded
- Total memory proportional to

$$\frac{1}{\varepsilon^2} \ln \frac{1}{\delta} (\log n + \log t)$$

Update time: $O(c)$ word operations

How do we get the “good” hash functions?

- Solution 1: Generate $b_1, \ldots, b_n$ at random once, store them
  - n bits, too much

- Solution 2: E.g., linear congruential method: $f(x) = a \cdot x + b$
  - OK if $a, b \leq n$, so $O(\log n)$ bits to store
  - But: h far from random: given $h(x)$, $h(y)$, get a, b by solving

$$h(x) = ax + b$$
$$h(y) = ay + b$$

Reducing Randomness

Reducing Randomness

Where did we use independence of the b_i's, really? For example, here:

$$E[b_i b_j] = E[b_i] \cdot E[b_j] = 0$$

For this, it is enough to have *pairwise independence*:

$$\text{For every } i, j, \quad \Pr[A_i | A_j] = \Pr[A_i]$$

Much weaker than full independence:

$$\text{For every } i, j, \quad \Pr[A_i | A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m] = \Pr[A_i]$$

Generating Pairwise Independent Bits

Choose f at random from a "small" family of pairwise independent functions

- $f(x)$, $f(y)$ guaranteed to be pairwise independent
- Each f in the family can be stored with $O(\log n)$ bits

Generating Pairwise Independent Bits (details)

- Work over finite field of size $q \simeq n$ (say q prime or $q = 2^r$)
- Idea: Choose $a, b \in [q]$ at random. Let $f(x) = a \cdot x + b$
- $2 \log q$ bits to store f
- Study system of equations

$$ax + b = \alpha, \qquad ay + b = \beta$$

- Given x, y ($x \neq y$!), α, β, exactly one solution for a, b
- Therefore, $\Pr_f[f(x) = \alpha | f(y) = \beta] = \Pr_f[f(x) = \alpha] = 1/q$
- Likewise: There are families of k-wise independent hash functions that can be stored in $k \log q \simeq k \log n$ bits

Completing the proof

- The proof of Claim 3 (bound on $Var(S \cdot T)$) needs 4-wise independence
- Algorithm initially chooses a random hash function f in a 4-wise independent family
- Remembers it using $4 \log n$ bits
- Each time it needs b_i, it computes $(-1)^{f(i) \bmod 2}$

About Pairwise Independence

Exercise 1

Verify that for pairwise independent variables X_i with $Var(X_i) = \sigma^2$ we have

$$Var(\frac{1}{k}\sum_{i=1}^{k} X_i) = \frac{\sigma^2}{k}$$

So: to reduce variance at a Chebyshev rate $1/k$ by averaging k copies, pairwise independence

To have a Hoeffding-like rate $\exp(-c\,k)$ we need full independence

Applications

- Computing L_2-distance

$$L_2(u,v) = \sum_{i=1}^{n} (u[i] - v[i])^2 = IP(u-v, u-v)$$

- Computing second frequency moment:

$$F_2 = \sum_{i=1}^{n} f_i^2 = IP(f,f)$$

Computing frequency moments

Frequency Moments

- k-th frequency moment of the sequence:

$$F_k = \sum_{i=1}^{n} f_i^k$$

- $F_0 =$ number of distinct symbols occurring in S
- $F_1 =$ length of sequence
- $F_2 =$ inner product of f with itself
- Define

$$F_\infty = \lim_{k\to\infty} (F_k)^{1/k} = \max_{i=1}^{n} f_i$$

Computing moments

[AMS] Noga Alon, Yossi Matias, Mario Szegedy (1996):
“The space complexity of approximating the frequency moments”

- Considered to initiate “data stream algorithmics”
- Studied the complexity of computing moments F_k
- Proposed approximation, proved upper and lower bounds
- Starting point for a large part of future work

Frequency Moments

$$F_k = \sum_{i=1}^{n} f_i^k$$

- Obvious algorithm: One counter per symbol. Memory $n \log t$
- [AMS] and many other papers, culminating in [Indyk, Woodruff 05]

- For $k > 2$, F_k can be approximated with $\widetilde{O}(n^{1-2/k})$ memory
- This is optimal. In particular, F_∞ requires $\Omega(n)$ memory
- For $k \leq 2$, F_k can be approximated with $O(\log n + \log t)$ memory
- Dependence is $\widetilde{\theta}(\varepsilon^{-2} \ln(1/\delta))$ for relative approximation

Counting distinct elements

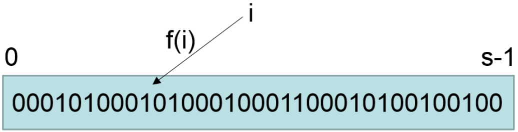
Counting distinct elements

Given a stream of elements from $[n]$, approximate how many distinct ones d have we seen at any time t

There are linear and logarithmic memory solutions
(in $d_{\max} \leq n$ if known a priori)

[Metwaly+08] good overview

Linear counting [Whang+90] $\simeq$ Bloom filters



Init:

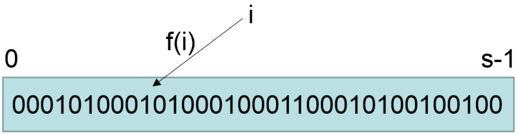
- choose a hash function $h : [n] \to s$;
- choose load factor $0 < \rho \leq 12$;
- build a bit vector B of size $s = d_{\max}/\rho$

Update(x): $B[h(x)] \leftarrow 1$

Query:

- w = the fraction of 0's in B;
- return $s \cdot \ln(1/w)$

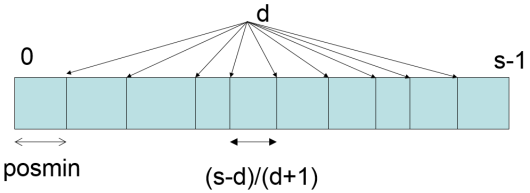
Linear counting [Whang+90] $\simeq$ Bloom filters



w = Prob[a fixed bucket is empty after inserting *d* distinct elements] = $(1-1/s)^d \simeq \exp(-d/s)$

$$E[\text{Query}] \simeq d, \qquad \sigma(\text{Query}) = \text{small!}$$

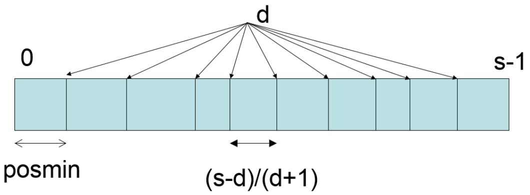
Cohen's algorithm [Cohen97]



E[gap between two 1's in B] = $(s - d)/(d + 1) \simeq s/d$

Query: return s / (size of first gap in B)

Cohen's algorithm [Cohen97]



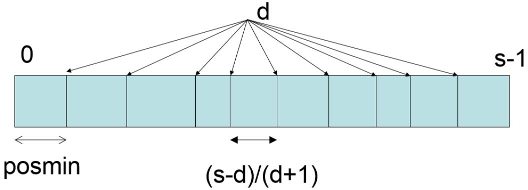
Trick: Don't store B, remember smallest key inserted in B

Init: posmin = s; choose hash function $h : [n] \to s$

Update(x): if ($h(x) <$ posmin) posmin $\leftarrow h(x)$

Query: return $s/$posmin;

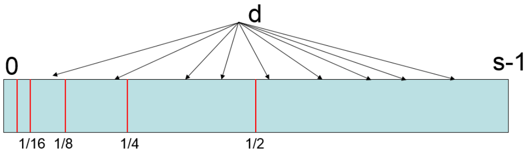
Cohen's algorithm [Cohen97]



$E[posmin] \simeq s/d \qquad \sigma(posmin) \simeq s/d$

Space is $\log s$ plus space to store h, i.e. $O(\log n)$

Probabilistic Counting



Flajolet-Martin counter [Flajolet+85]
+ LogLog + SuperLogLog + HyperLogLog

Observe the values of $f(i)$ where we insert, in binary

Idea: To see $f(i) = 0^{k-1}1\ldots$, 2^k distinct values inserted

And we don't need to store B; just the smallest k

Flajolet-Martin probabilistic counter

Init: $p = \log n$;

Update(x):

- let b be the position of the leftmost 1 bit of $h(x)$;
- if $(b < p)$ $p \leftarrow b$;

Query: return 2^p;

$E[2^p] =$ number of distinct elements
Space: $\log p = \log \log n$ bits

Flajolet-Martin: reducing the variance

Solution 1: Use k independent copies, average

- Problem: runtime multiplied by k
- Problem: now pairwise independent hash functions don't seem to suffice
- We don't know how to generate several fully independent hash functions

In fact, we don't know how to generate one fully independent hash functions
But good quality crypto hash functions work in this setting - even weaker ones ("2-universal hash functions") with a minimum of entropy. And use $O(\log n)$ bits

Flajolet-Martin: reducing the variance

Solution 2:

- Divide stream into $m = O(\varepsilon^{-2})$ substreams
- Use first bits of $h(x)$ to decide substream for x
- Track p separately for each substream
- Now a single h can be used for all copies
- One sketch updated per item

- Query: Drop top and bottom 20% of estimates, average the rest

Space: $O(m \log\log n + \log n) = O(\varepsilon^{-2} \log\log n + \log n)$

Improving the leading constants

- SuperLogLog [Durand+03]: Take geometric mean
- HyperLogLog [Flajolet+07]: Take harmonic mean

"cardinalities up to 10^9 can be approximated within say 2% with 1.5 Kbytes of memory"

[Kane+10] Optimal $O(\varepsilon^{-2} + \log n)$ space, $O(1)$ update time