Data Streams

Everything Data

CompSci 216 Spring 2015

DUKE
COMPUTER SCIENCE

Announcements (Wed. Apr. 8)

- **Homework #12** to be posted by tomorrow
- **Project mid-term feedback** to be emailed by this weekend

- **T-shirt design contest**: see email for details

Data stream

- A **potentially infinite sequence**, where data arrives one record at a time
- We only get **one look**—can't go back
- We want to answer a **standing query** that produces new/updated results as stream goes by
- We have **limited space** to remember whatever we deem necessary to answer the query

Several questions about streams

How do we maintain a random sample of size *n* for all data we've seen so far?

– Sample can be used to answer queries

How do we maintain a data structure to check if a new arrival has appeared before?

– E.g., a URL shortening service

How do we count the number of unique records seen so far?

– E.g., # of unique visitors (by IP) to a website

Sampling static data vs. stream

- With a static dataset
  - We know the total data size N
  - We can access an arbitrary record
- With a stream
  - There is no N, just the number of records we have previously seen
  - We only get one look of any record, in arrival order

Reservoir sampling

- Make one pass over data
- Maintain a reservoir of n records
- After reading t records, the reservoir is a random sample of the first t records

- The algorithm tells us how to update the reservoir upon every new record arrival

Simple algorithm

- Initialize reservoir to the first n records
- For the t-th (new) record
  - Pick a random number x between 1 and t
  - If $x \leq n$, then replace the x-th record in the reservoir with the new record
- That's it!

But why?

- If $t = n$, obviously the reservoir has a "random sample" of all records seen so far
- Suppose the reservoir is a random sample of the first t records for $t = k - 1$
  - I.e., P[r in reservoir after $k - 1$ steps] = $n/(k - 1)$
- What happens when $t = k$?
  - The new record is included with prob. n / k
  - For any other record r, P[r in reservoir]

= P[r in reservoir after $k - 1$ steps]

× P[r is not replaced in step k]

= $[n/(k - 1)] \times (1 - 1/k) = n / k$

An improvement

What if skipping new records is cheaper than accessing them one by one?

After adding the *t*-th record to reservoir…

- Simulate forward until we need to add a new record in the reservoir; skip until then
- Or calculate the CDF of skip size
  P[skip size ≤ s] = $1-\left(\frac{t+1-n}{t+1}\right)\left(\frac{t+2-n}{t+2}\right)\cdots\left(\frac{t+s+1-n}{t+s+1}\right)$
  - Sample from this CDF, skip accordingly
  - Pick one record in reservoir to replace

Summary of reservoir sampling

- Helps create a “running” random sample of fixed size over a stream

- Very useful when computing/accessing the whole dataset is expensive

Outline

How do we maintain a random sample of size n for all data we've seen so far?

- Sample can be used to answer queries

How do we maintain a data structure to check if a new arrival has appeared before?

- E.g., a URL shortening service

How do we count the number of unique records seen so far?

- E.g., # of unique visitors (by IP) to a website

Problem boils down to…

- Give a large set S (e.g., all values seen so far), check whether a given value x is in S
- Suppose we have n bits of storage ($n \ll |S|$)
  - Cannot afford to store S

Approximation comes to rescue

- If x is in S, return *true* with prob. 1
  - I.e., no false negatives
- If x is not in S, return *false* with high prob.
  - I.e., possible false positives

Primitive: hash function

$h: S \longrightarrow \{1, 2, \ldots, n\}$

- Hashes values uniformly to integers in $[1, n]$, i.e.: $P[h(x) = i] = 1/n$

- "Compressing" a value down with one h loses too much information, so we use k *independent* hash functions $h_1, h_2, \ldots, h_k$

Bloom filter

Initialization

- Set all n bits to 0

Add x to S

- Compute $h_1(x)$, $h_2(x)$, …, $h_k(x)$
- Set the corresponding k bits to 1

Check if x is in S

- Compute $h_1(x)$, $h_2(x)$, …, $h_k(x)$
- Return *true* iff the corresp. k bits are all 1

No false negatives

If x is really in S

- Then by construction we have set bits $h_1(x)$, $h_2(x)$, …, $h_k(x)$ to 1
- So check will surely return *true*

False positive probability

If x is not in S

- Check returns *true* if each bit $h_j(x)$ is 1 due to some other value(s) in S
- P[bit i is 1]
  $= 1 - $ P[bit i was not set by $k|S|$ hashes]
  $= 1 - (1 - 1/n)^{k|S|}$
- P[k particular bits are 1]
  $= (1 - (1 - 1/n)^{k|S|})^k$
  $\approx (1 - e^{-k|S|/n})^k$

Example

- Suppose there are $|S| = 10^9$ elements
- Suppose we have 1 GB (8×10^9 bits) memory

If $k = 1$

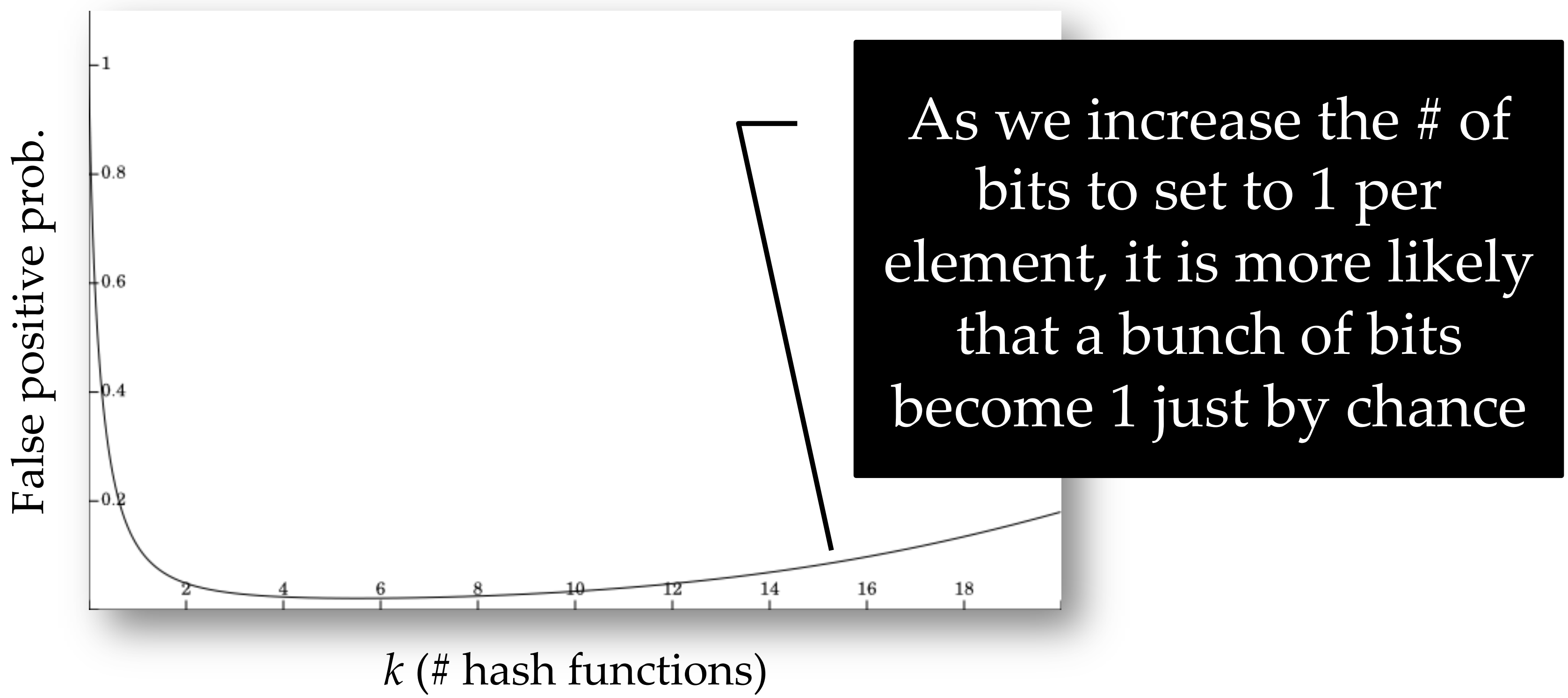
- P[false positive] $\approx (1 - e^{-k|S|/n})^k = 1 - e^{-1/8}$
  ≈ 0.1175

If $k = 2$

- P[false positive] $\approx (1 - e^{-k|S|/n})^k = (1 - e^{-2/8})^2$
  ≈ 0.0493

Example

- Suppose there are $|S| = 10^9$ elements
- Suppose we have 1 GB (8×10^9 bits) memory



Summary of Bloom filter

- Helps check membership in a large set that cannot be stored entirely
- No false negatives
  - Good for applications like URL shortener
- False negative probability can be tweaked by the choice of n and k

Outline

How do we maintain a random sample of size n for all data we've seen so far?

- Sample can be used to answer queries

How do we maintain a data structure to check if a new arrival has appeared before?

- E.g., a URL shortening service

How do we count the number of unique records seen so far?

- E.g., # of unique visitors (by IP) to a website

Can you use a Bloom filter?

- Increment a counter whenever check returns false for an incoming value
- Because of 0 false negative and non-0 false positive probabilities, we will consistently underestimate the # of distinct values
- Also, the Bloom filter does more than we need—can we use the n bits more efficiently?

FM (Flajolet-Martin) sketch

Let $\text{Tail}_0(h(x))$ = # of trailing consecutive 0's

- $\text{Tail}_0(101001) = 0$
- $\text{Tail}_0(101010) = 1$
- $\text{Tail}_0(001100) = 2$
- $\text{Tail}_0(101000) = 3$
- $\text{Tail}_0(000000) = 6$

FM sketch

- Maintain a value K (max 0-tail length)
- Initialize K to 0
- For each new value
  - Compute $\text{Tail}_0(h(x))$
  - Replace K with this value it is greater than K
- $F' = 2^K$ is an estimate of F, the true number of distinct elements

- K require very little space to store

Rough intuition

If we have F distinct elements, we'd expect

- $F/2$ of them to have $\text{Tail}_0(x) = 0$
- $F/4$ of them to have $\text{Tail}_0(x) = 1$
- …
- $F/2^i$ of them to have $\text{Tail}_0(x) = i$
- …

So $F' = 2^K$ is pretty good guess of F

How good is the result?

- F: the true number of distinct elements
- F': guess by FM sketch
- We can show that for all $c > 3$,

$$P[F/c \leq F' \leq cF] > 1 - 3/c$$

- But that's not very accurate!

Use more sketches!

- Use the "median of means" trick
- Maintain $a \times b$ FM sketches
  - Use *independent* hash functions!
- Compute the mean over each group of a
- Return the median of b means as answer

Summary of FM sketch

- Helps estimate # of distinct elements in a large set that cannot be stored entirely
- Each FM sketch is very rough, but groups of them improve estimation
- Trick question: do FM sketches support membership check like Bloom filter?
  - No—too much error on any particular check
  - Specialization gives us better efficiency

Summary

Tricks for big data covered in class

- Parallel processing (e.g., MapReduce)
- Approximate processing
  - Sampling (downsize data)
  - Stream processing (linear time, limited space)

Image: http://bigdatapix.tumblr.com/