Mining Data Streams (Part 2)

A. Farzindar
farzinda@usc.edu

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman Stanford University http://www.mmds.org

Today's Lecture

- More algorithms for streams:
  - Sampling data from a stream
  - Filtering a data stream: Bloom filters
  - Counting distinct elements: Flajolet-Martin
  - Estimating moments: AMS method.

Sampling from a data stream

- Method 1: sample a fixed portion of elements
  - e.g., 1/10
- Method 2: maintain a fixed-size sample.

Sampling from a Data Stream: Sampling a fixed proportion

- **As the stream grows the sample also gets bigger**

Sampling from a Data Stream

- Since **we can not store the entire stream**, one obvious approach is to store a **sample**
- **Two different problems:**
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - At any "time" ***k*** we would like a random sample of ***s*** elements
      - **What is the property of the sample we want to maintain?**
        For all time steps ***k***, each of ***k*** elements seen so far has **equal prob. of being sampled.**

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**
- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as:** **How often did a user run the same query in a single days**
  - Have space to store **1/10th** of query stream
- **Naïve solution:**
  - Generate a **random integer** in **[0..9]** for each query
  - Store the query if the integer is **0**, otherwise discard.

Example: Unique search queries

- The length of the sample is **10% of the length** of the whole stream
- Suppose a query is **unique**
  - It has a **10% chance** of being in the sample
- Suppose a query **occurs exactly twice** in the stream
  - It has an **18% chance** of appearing exactly once in the sample.
    - **$(1/10 \cdot 9/10)+(9/10 \cdot 1/10)=0.18$**
- And so on … The fraction of unique queries in the stream is unpredictably large.

Problem with Naïve Approach

- **Simple question: What fraction of queries by an average search engine user are duplicates?**
  - Suppose each user issues ***x* queries once** and ***d* queries twice** (total of ***x*+2*d*** queries)
    - **Correct answer: *d*/(*x*+*d*)**
  - **Proposed solution: We keep 10% of the queries**
    - Sample will contain ***x*/10** of the **singleton queries** and **2*d*/10** of the **duplicate queries** at least once
    - But only ***d*/100** **pairs of duplicates**
      - **d/100 = 1/10 · 1/10 · d**
    - Of ***d*** "duplicates" ***18d/100*** **appear exactly once**
      - **18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d**
        - **d_j selected, d_j' not selected: 1/10 * 9/10**
        - **d_j not selected, d_j' selected: 9/10 * 1/10**
  - **So the sample-based answer is** $\frac{\frac{d}{100}}{\frac{x}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10x+19d} \neq d/(x+d)$

Solution: Sample Users

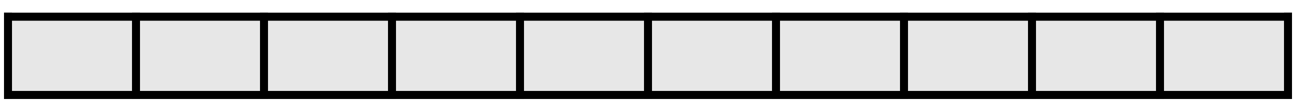
Our mistake: we sampled based on the **position** in the stream, rather than **the value of the stream element**

Solution:

- Pick **1/10th** of **users** and take all their searches in the sample
- Use a hash function that hashes the user name or user id uniformly into 10 buckets.

Generalized Solution

- **Stream of tuples with keys:**
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search, time); key is **user**
  - Choice of key depends on application
- **To get a sample of *a/b* fraction of the stream:**
  - Hash each tuple's key uniformly into ***b*** buckets
  - Pick the tuple if its hash value is at most ***a***



Hash table with **b** buckets, pick the tuple if its hash value is at most **a.**
How to generate a 30% sample?
Hash into **b=10 buckets**, take the tuple if it hashes to one of the **first 3 buckets**

Sampling from a Data Stream: Sampling a fixed-size sample

- **As the stream grows, the sample is of fixed size**

Problem with fixed portion sample

- Sample size may grow too big when data stream in
  - Even 10% could be too big
- Idea: throw away some queries
- Key: do this consistently
  - remove all or none of occurrences of a query.

Controlling the sample size

- Put an upper bound on the sample size
  - Start out with 10%

- Solution:
- Hash queries to a large # of buckets, say 100
  - Take for the sample those elements hashing to buckets 0 through 9.
- When sample grows too big, throw away bucket 9
- Still too big, get rid of 8, and so on.

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first s elements of the stream to S
  - Suppose we have seen n-1 elements, and now the n^{th} element arrives ($n > s$)
    - With probability s/n, keep the n^{th} element, else discard it
    - If we picked the n^{th} element, then it replaces one of the s elements in the sample S, picked uniformly at random
- **Claim:** This algorithm maintains a sample S with the desired property:
  - After n elements, the sample contains each element seen so far with probability s/n.

Filtering Data Streams

Filtering Data Streams

- **Each element of data stream is a tuple**
- Given a list of keys **S**
- **Determine which tuples of stream are in *S***

- **Obvious solution: Hash table**
  - But suppose we **do not have enough memory** to store all of ***S*** in a hash table
    - E.g., we might be processing millions of filters on the same stream.

Applications

- **Example: Email spam filtering**
  - We know 1 billion "good" email addresses
  - If an email comes from one of these, it is **NOT** spam
- **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user's interest.

Filtering Stream Content

- Consider a **web crawler**
- It keeps a **list of all the URL's** it has found so far
- It assigns these URL's to any of a number of parallel tasks;
  - these tasks stream back the URL's they find in the links they discover on a page
- It needs to filter out those URL's it **has seen before.**

Role of the Bloom Filter

- A Bloom filter placed on the stream of URL's will declare that **certain URL's have been seen before**
- Others will be **declared new**, and **will be added** to the list of URL's that need to be crawled
- Unfortunately, the Bloom filter can have **false positives**
  - It can declare a **URL has been seen before** when it hasn't
  - But if it says **"never seen," then it is truly new (no False Negative)**.

How a Bloom Filter Works

- A ***Bloom filter*** is an array of bits, together with a number of **hash functions**
- The argument of each hash function is a **stream element**, and it returns a **position** in the array
- Initially, all bits are 0
- When input x arrives, we set to 1 the bits h(x),
- for each hash function h.

Example: Bloom Filtering

- Use N = 11 bits for our filter
- Stream elements = integers
- Use two hash functions:
  - h1(x) =
    - Take **odd numbered** bits from the **right** in the **binary representation** of x
    - Treat it as an integer i
    - Result is i modulo 11
- h2(x) = same, but take **even numbered** bits.

Example: Building the filter

h1(x) = **odd position** bits from the **right**
h2(x) = **even position**

Stream element	h_1	h_2	Filter
			0 0 0 0 0 0 0 0 0 0 0
25 = 1 1 0 0 1	5	2	0 0 1 0 0 1 0 0 0 0 0
159 = 1 0 0 1 1 1 1 1	7	0	1 0 1 0 0 1 0 1 0 0 0
585 = 1 0 0 1 0 0 1 0 0 1	9	7	1 0 1 0 0 1 0 1 0 1 0

Bloom Filter Lookup

- Suppose element **y appears** in the stream, and we want to know **if we have seen y before**
- **Compute h(y)** for each hash function y
- If all the resulting **bit positions are 1**, say we have **seen y before** **(false positive)**
- If at least one of these **positions is 0**, say we have **not seen y before** **(false negative)**.

Example: Lookup

- Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010.
- Lookup element **y = 118** = 1110110 (binary).
- h1(y) = 14 modulo 11 = 3.
- h2(y) = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are **sure y is not in the set.**

Performance of Bloom Filters

- **Probability of a false positive** depends on the **density of 1's** in the array and the **number of hash functions**
  - = (fraction of 1's)# of hash functions
- The number of 1's is approximately the number of elements inserted times the number of hash functions
  - But collisions lower that number slightly.

Analysis: Throwing Darts (1)

- Turning random bits from 0 to 1 is like throwing ***d*** darts at ***t*** targets, at random
- **More accurate analysis for the number of false positives**
- **Consider:** If we throw ***d*** darts into ***t*** equally likely targets, **what is the probability that a target gets at least one dart?**
- **In our case:**
  - **Targets** = bits/buckets
  - **Darts** = hash values of items.

Analysis: Throwing Darts (2)

- We have ***d*** darts, ***t*** targets
- **What is the probability that a target gets at least one dart?**

- Probability a given target is hit by a given dart

  $$= 1/t$$

- Probability non of d darts hit a given target is

  $$=(1 - 1/t)^{d}$$

Analysis: Throwing Darts (2)

- We have ***d*** darts, ***t*** targets
- **What is the probability that a target gets at least one dart?**

Equals **1/e** as **n** $\rightarrow\infty$

Equivalent

$$1 - (1 - 1/t)^{t\left(\frac{d}{t}\right)}$$

$$1 - e^{-d/t}$$

Probability some target **X** not hit by a dart

Probability at least one dart hits target **X**

Example: Throwing Darts

- **Fraction of 1s in the array B =**
  = probability of false positive = 1 – $e^{-d/t}$
- Example: Suppose we use an array of **1 billion bits**, **5 hash functions**, and **we insert 100 million** elements
- That is, **$t = 10^9$**, and d = **$5 * 10^8$**
  - The **fraction of 0's** that remain will be $e^{-1/2} = 0.607$
  - **Density of 1's** = 0.393
- Probability of a false positive = $(0.393)^{5}$ = 0.00937.

Summary

- More algorithms for streams:
  - Sampling data from a stream
    - Method 1: sample a fixed portion of elements
    - Method 2: maintain a fixed-size sample
  - Filtering a data stream: Bloom filters
  - Counting distinct elements: Flajolet-Martin
  - Estimating moments: AMS method.