Mining Data Streams (Part 1)

Zainab Abbas and Vladimir Vlassov

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Outline

- Part 1 (this lecture)
  - The stream data model
  - Sampling data in a stream
  - Filtering streams
- Part 2 (next lecture: counting algorithms on streams)
  - Counting distinct elements
  - Computing moments
  - Queries over a (long) sliding window: Counting ones
  - Decaying Windows: Counting itemsets

Data Streams – Infinite Data

In many data mining situations, we do not know the entire data set in advance

Stream Management is important when the input rate is controlled **externally:**

- Google search queries
- Twitter or Facebook status updates

We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

The Stream Model

Input **elements** enter at a rapid rate, at one or more input ports (i.e., ***streams***)

– **We call elements of the stream tuples**

The system cannot store the entire stream accessibly

Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?

SGD – Example of a Streaming Algorithm

Stochastic Gradient Descent (SGD) is an example of a stream algorithm
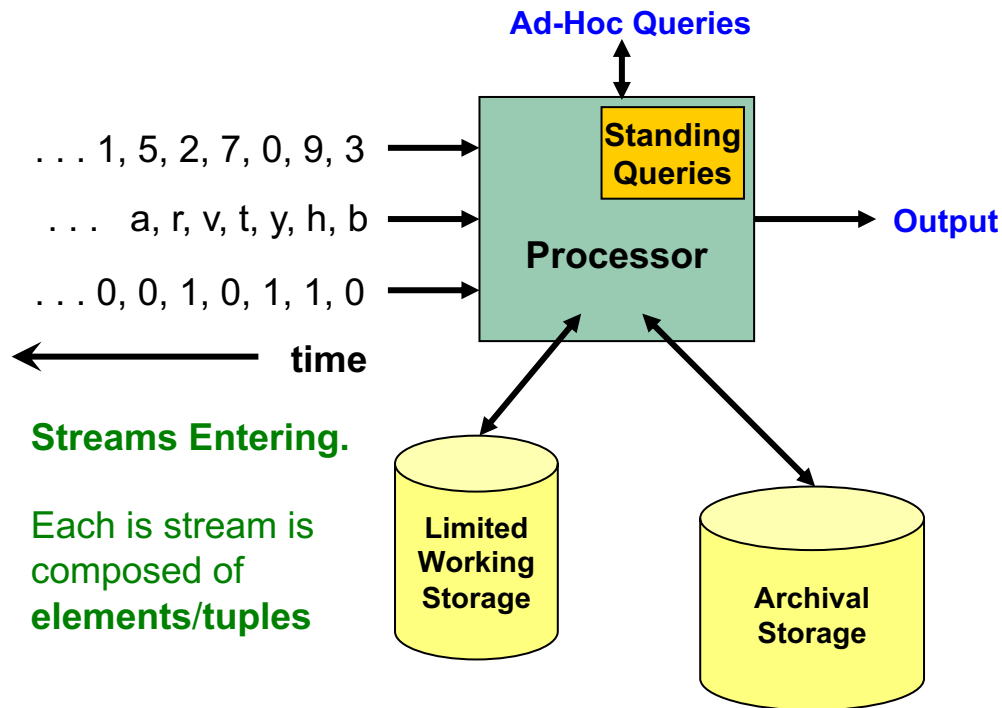
In Machine Learning we call this: Online Learning

– Allows for modeling problems where we have a continuous stream of data

– We want an algorithm to learn from it and slowly adapt to the changes in data

Idea: Do slow updates to the model

– **SGD** (SVM, Perceptron) makes small updates

– **So:** First train the classifier on training data.

– **Then:** For every example from the stream, we slightly update the model (using small learning rate)

General Stream Processing Model

. . . 1, 5, 2, 7, 0, 9, 3

. . . a, r, v, t, y, h, b

. . . 0, 0, 1, 0, 1, 1, 0

time

Streams Entering.

Each is stream is composed of **elements**/**tuples**

Ad-Hoc Queries

Standing Queries

Processor

Output

Limited Working Storage

Archival Storage

Two forms of query

1. ***Ad-hoc queries***: Normal queries asked one time about streams.
   - **Example**: What is the maximum value seen so far in stream *S*?

2. ***Standing queries***: Queries that are, in principle, asked about the stream at all times.
   - **Example**: Report each new maximum value ever seen in stream S.

Problems on Data Streams (1)

Types of queries one wants on answer on a data stream:

(to be considered in this lecture)

- **Sampling data from a stream**
    - Construct a random sample

- **Filtering a data stream**
    - Select elements with property x from the stream

Problems on Data Streams (2)

Types of queries one wants on answer on a data stream: (to be considered in next lecture)

- **Queries over sliding windows**
  - Number of items of type x in the last k elements of the stream

- **Counting distinct elements**
  - Number of distinct elements in the last k elements of the stream

- **Estimating moments**
  - Estimate avg./std. dev. of last k elements

- **Finding frequent elements**

Applications (1)

Mining query streams

- Google wants to know what queries are more frequent today than yesterday

Mining click streams

- Yahoo! wants to know which of its pages are getting an unusual number of hits in the past hour

  – Often caused by annoyed users clicking on a broken page.

Mining social network news feeds

- E.g., look for trending topics on Twitter, Facebook

Applications (2)

Sensor Networks

- Many sensors feeding into a central controller

Telephone call records

- Data feeds into customer bills as well as settlements between telephone companies

IP packets monitored at a switch

- Gather information for optimal routing
- Detect denial-of-service attacks

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 probability 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

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, i.e., in total of $(x + 2d)$ queries

  - **Correct answer is** $\boxed{d/(x+d)}$

- **Proposed solution: Let's keep 10% of the queries**

  - Sample will contain $x/10$ of the singleton queries and $2d/10$ of the duplicate queries at least once

  - But only $d/100$ pairs of duplicates: $d/100 = 1/10 \cdot 1/10 \cdot d = 0{,}01d$

  - Of d "duplicates" $18d/100$ appear exactly once: $((1/10 \cdot 9/10)+(9/10 \cdot 1/10)) \cdot d = 0{,}18d$

- **So the sample-based answer is** $\dfrac{0{,}01d}{0{,}1x + 0{,}01d + 0{,}18d} = \dfrac{d}{10x + 19d}$

Solution: Sample Users

- 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 _a_=3 buckets

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

As the stream grows, the sample is of a fixed size

Maintaining a Fixed-size Sample

Problem 2: Fixed-size sample

Suppose we need to maintain a random sample S of size exactly s tuples

- E.g., main memory size constraint

Why? Don't know length of stream in advance

Suppose at time n we have seen n items

- **Each item is in the sample S with equal prob. s/n**

Maintaining a Fixed-size Sample (cont)

How to think about the problem: say the sample size $s = 2$

Stream: a x c y z k c d e g…

At **$n= 5$,** each of the first 5 tuples is included in the sample **S** with equal probability.

At **$n= 7$,** each of the first 7 tuples is included in the sample **S** with equal probability.

Impractical solution would be to store all the n tuples seen so far and out of them pick s at random

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**

Proof: By Induction

We prove this by induction:

- Assume that after n elements, the sample contains each element seen so far with probability s/n

- We need to show that after seeing element $n+1$ the sample maintains the property

  – Sample contains each element seen so far with probability $s/(n+1)$

Base case:

- After we see $n=s$ elements the sample S has the desired property

  – Each out of $n=s$ elements is in the sample with probability $s/s = 1$

Proof: By Induction (cont)

Inductive hypothesis: After n elements, the sample S contains each element seen so far with probability s/n

Now element $n+1$ arrives

Inductive step: For elements already in S, probability that the algorithm keeps it in S:

$$\left(1 - \frac{s}{n+1}\right) + \left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right) = \frac{n}{n+1}$$

Element $n+1$ discarded   Element $n+1$ not discarded   Element in the sample not picked

- So, at time n, tuples in S were there with probability s/n
- At time $n \rightarrow n+1$, tuple stayed in S with prob. $n/(n+1)$
- So prob. tuple is in S at time $n+1$: $(s/n) \cdot (n/n+1) = \frac{s}{n+1}$

Filtering Data Streams

Filtering Data Streams

Another common process on streams is *selection*, or *filtering*.

- Each element of data stream is a tuple
- Tuples in the stream that meet a criterion are accepted.
  - Accepted tuples are passed to another process as a stream
  - Non-accepted tuples are dropped.

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: **Store keys in a 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

First Cut Solution (1)

Given a set of keys *S* that we want to filter

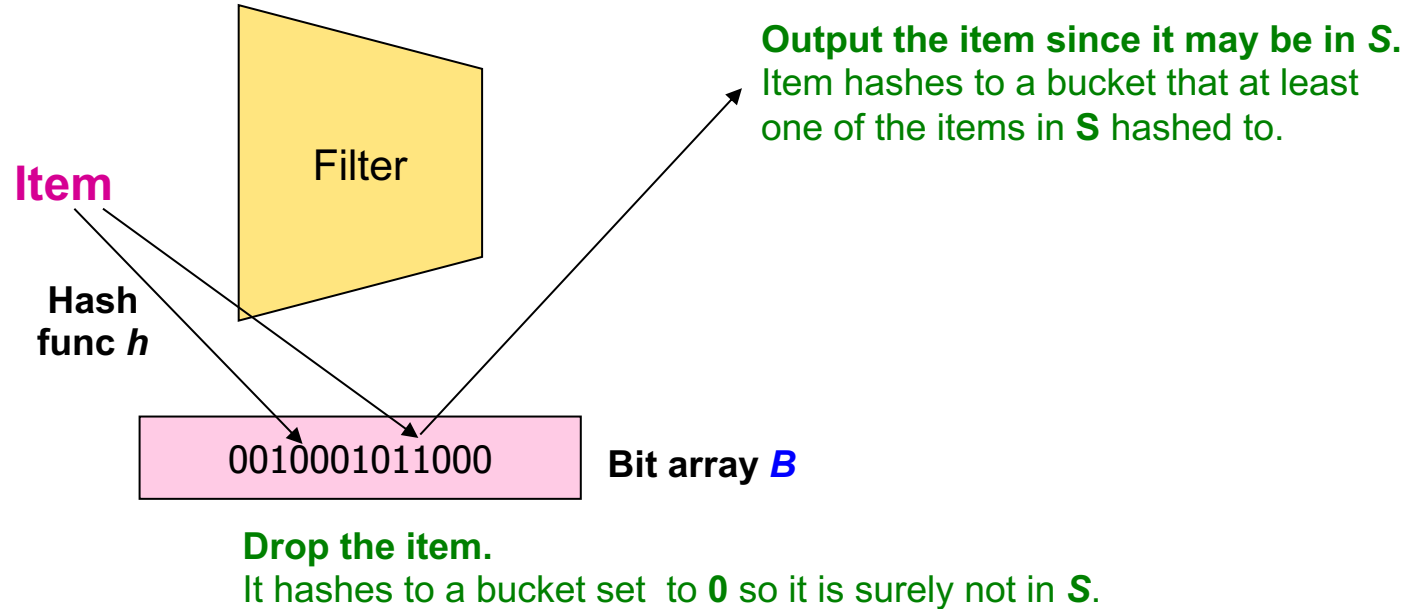Create a **bit array *B*** of ***n*** bits, initially all ***0*s**

Choose a **hash function *h*** with range **[0,*n*)**

Hash each member of ***s*∈ *S*** to one of ***n*** buckets, and set a corresponding bit to **1**, i.e., ***B*[*h*(*s*)]=1**

Hash each element ***a*** of the stream and output only those that hash to a bucket with its corresponding bit in B that was set to **1**

- **Output *a* if *B*[*h*(*a*)] == 1**

First Cut Solution (2)

Item

Filter

Output the item since it may be in *S*.
Item hashes to a bucket that at least
one of the items in **S** hashed to.

**Hash
func *h***

`0010001011000`   **Bit array *B***

Drop the item.
It hashes to a bucket set to **0** so it is surely not in **S**.

Creates false positives but no false negatives

– If the item is in *S* we surely output it, if not we may still output it

First Cut Solution (3)

- **|S| = 1 billion email addresses**
  |B|= 1GB = 8 billion bits

If the email address is in **S**, then it surely hashes to a bucket that has the bit set to 1, so it always gets through (*no false negatives*)

Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in **S** get through to the output (*false positives*)
  - Actually, less than **1/8th**, because more than one address might hash to the same bit

Bloom Filter

A Bloom filter consists of:

1. An bit-array B of n bits, initially all 0's.

2. A collection of hash functions $h_1, h_2, . . . , h_k$. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.

3. A set S of m key values

The Bloom filter allows through all stream elements whose keys are in S

- It rejects stream elements whose keys are not in S

Bloom Filter (cont)

Initialization:

- Set **B** to all **0s**
- Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1,.., k$)

Run-time:

- When a stream element with key **x** arrives
  - If $B[h_i(x)] = 1$ <u>**for all**</u> $i = 1,..., k$ then declare that **x** is in **S**
    - That is, **x** hashes to a bucket set to **1** for every hash function $h_i(x)$
  - Otherwise discard the element **x**

Analysis of Bloom Filtering: Throwing Darts

Analysis for the number of false positives

(assuming one hash function $k = 1$)

Consider: If we throw m darts into n 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 (cont)

Assume, we have *x* targets and *y* darts

What is the probability that a target gets at least one dart?

The probability that a given dart will not hit a given target *x* is

$$(x - 1)/x$$

The probability that none of the *y* darts will hit a given target *x* is

$$\left(\frac{x-1}{x}\right)^y \quad = \quad \left(1 - \frac{1}{x}\right)^{x\left(\frac{y}{x}\right)} \quad \approx \quad e^{-y/x}$$

– using approximation $(1-\epsilon)^{1/\epsilon} = 1/e$

The probability that a target gets at least one dart: $1 - e^{-y/x}$

Bloom Filter – Analysis

Bloom filter: bit-array **B** of **n** bits; **k** hash functions; **m** keys

What fraction of the bit vector B are 1s?

- Throwing **k·m** darts at **n** targets
- So fraction of **1**s is $(1 - e^{-km/n})$

But we have **k** independent hash functions and
we only let the element **x** to pass through **if all k** hash functions hash
element **x** to a bucket of value **1**

So, false **positive probability** = $(1 - e^{-km/n})^k$
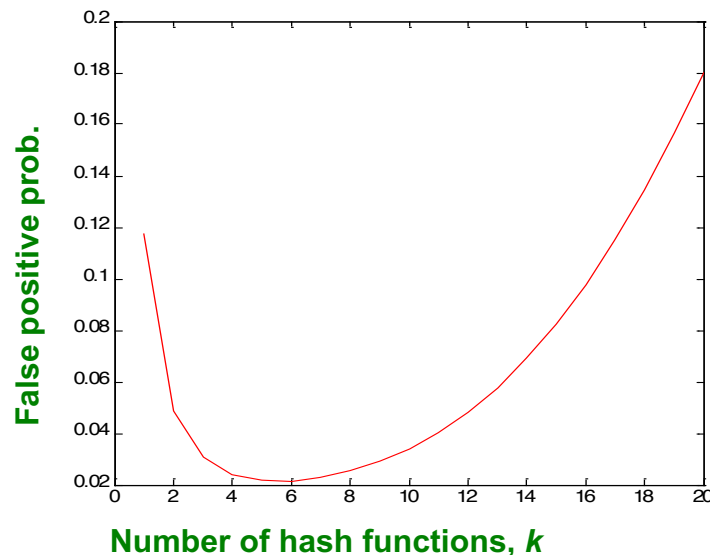
Bloom Filter – Analysis (cont)

m = 1 billion, n = 8 billion

- **k = 1**: $(1 - e^{-1/8}) = $ **0.1175**

- **k = 2**: $(1 - e^{-1/4})^2 = $ **0.0493**

What happens as we keep increasing k?

"Optimal" value of **k**: **n/m ln(2)**

- **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**

- **Error at k = 6**: $(1 - e^{-1/6})^2 = $ **0.0235**

False positive prob. vs. Number of hash functions, k

Bloom Filter: Wrap-up

Bloom filters guarantee no false negatives, and use limited memory

- Great for pre-processing before more expensive checks

Suitable for hardware implementation

- Hash function computations can be parallelized

Options: Is it better to have **one** big **B** or *k* small **B**s?

- **It is the same: $(1 - e^{-km/n})^k$** vs. $(1 - e^{-m/(n/k)})^k$
- But keeping **1 big B** is simpler