More on Data Streams

Shannon Quinn

(with thanks to J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org)

Data Streams

- **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 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**

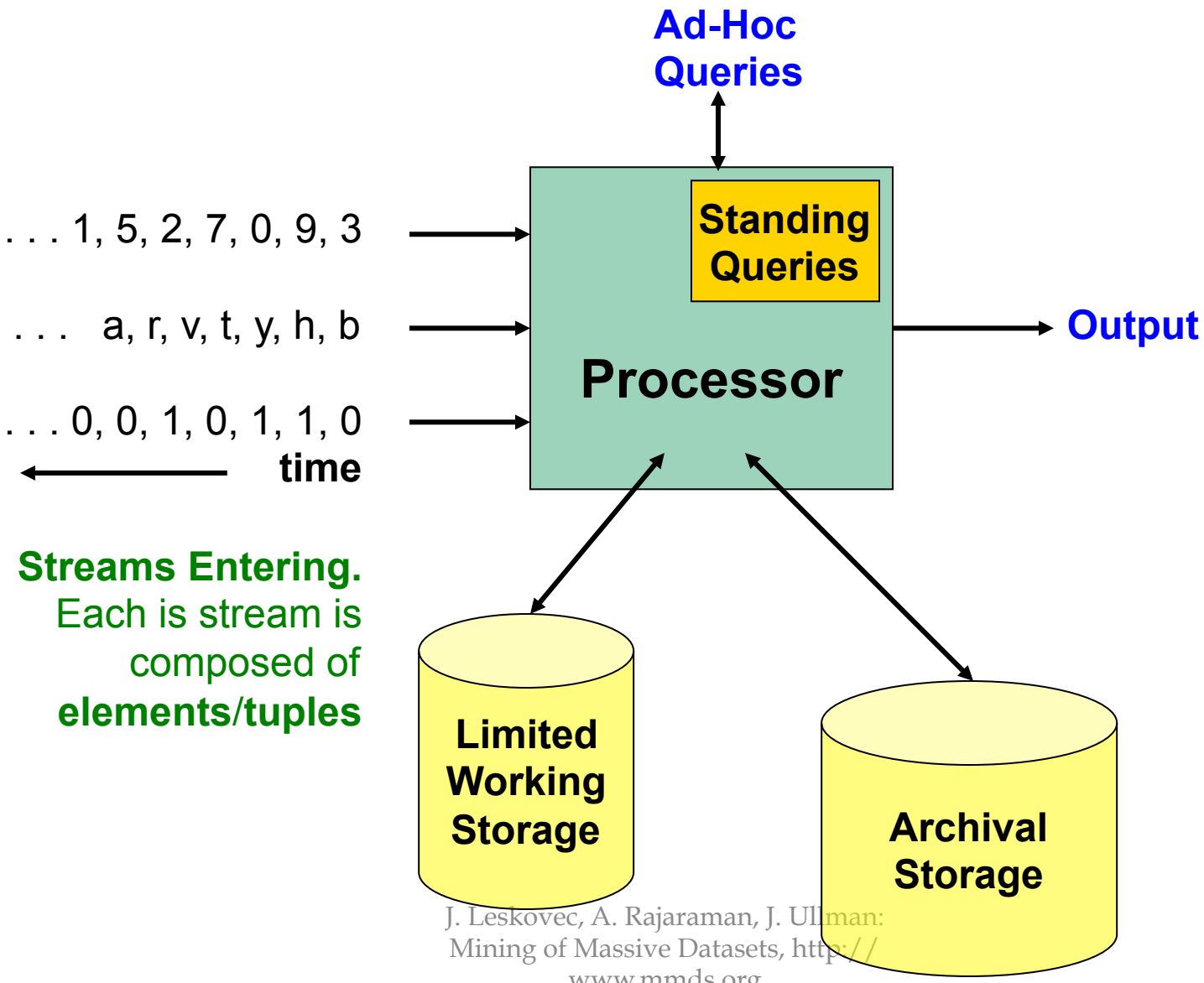
- **The system cannot store the entire stream accessibly**

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

Side note: NB is a Streaming Alg.

- **Naïve Bayes (NB) 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**
  - (NB, 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



Streams Entering. Each is stream is composed of **elements/tuples**

Problems on Data Streams

- **Types of queries one wants on answer on a data stream:** (we'll do these today)
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type x in the last k elements of the stream

Problems on Data Streams

- **Other types of queries one wants on answer on a data stream:**
  - **Filtering a data stream**
    - Select elements with property x from 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
- **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

- 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

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 $x+2d$ queries)
    - **Correct answer:** $d/(x+d)$
  - **Proposed solution: We 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
      - $d/100 = 1/10 \cdot 1/10 \cdot d$
    - Of d "duplicates" $18d/100$ appear exactly once
      - $18d/100 = ((1/10 \cdot 9/10)+(9/10 \cdot 1/10)) \cdot d$
  - **So the sample-based answer is**

Solution: Sample Users

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

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***

How to think about the problem: say 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 prob.

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

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

- **Inductive hypothesis:** After n elements, the sample S contains each element seen so far with prob. s/n
- **Now element $n+1$ arrives**
- **Inductive step:** For elements already in S, probability that the algorithm keeps it in S is:

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

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 prob. **s/n**
- Time $n \rightarrow n+1$, tuple stayed in S with prob. **n/(n+1)**
- So prob. tuple is in S at time $n+1$ =

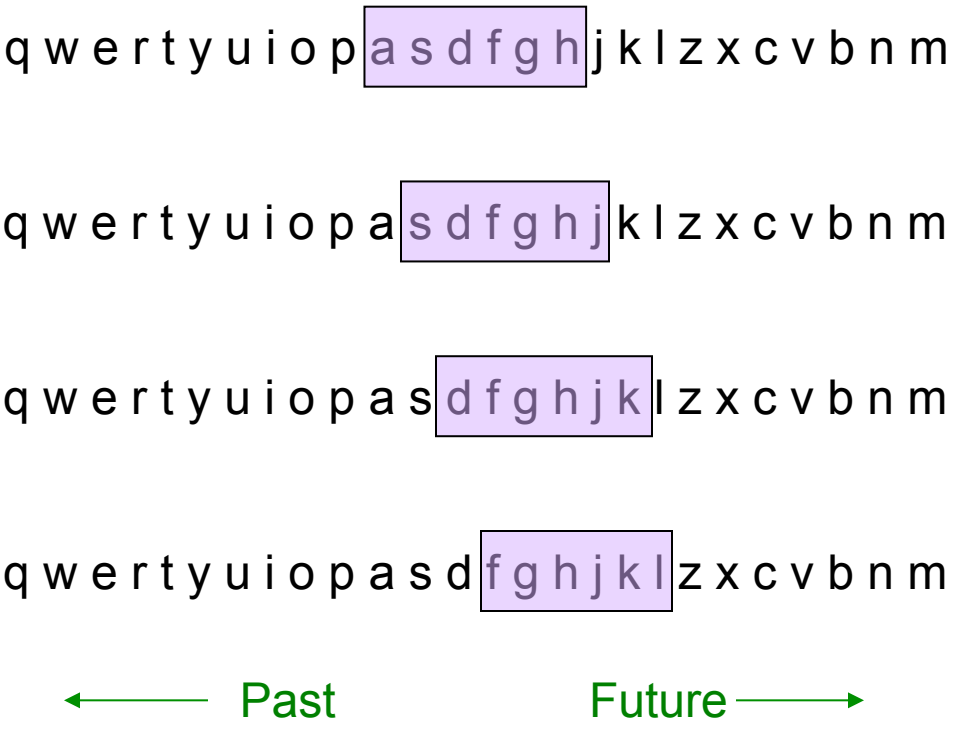
Sliding Windows

- A useful model of stream processing is that queries are about a *window* of length ***N*** – the ***N*** most recent elements received

- **Interesting case:** ***N*** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- **Amazon example:**
  - For every product **X** we keep 0/1 stream of whether that product was sold in the **n**-th transaction
  - We want answer queries, how many times have we sold **X** in the last **k** sales

Sliding Window: 1 Stream

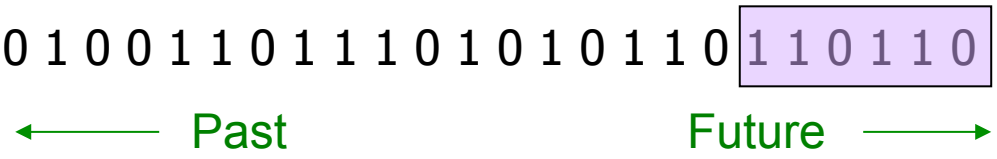
- **Sliding window on a single stream:** **N = 6**



Counting Bits (1)

- **Problem:**
  - Given a stream of **0**s and **1**s
  - Be prepared to answer queries of the form **How many 1s are in the last *k* bits?** where $k \leq N$

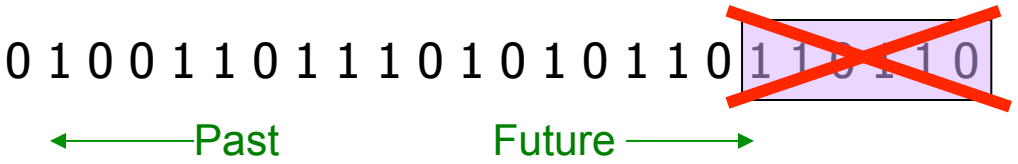
- **Obvious solution:**
  Store the most recent N bits
  - When new bit comes in, discard the N+**1**st bit



Counting Bits (2)

- **You can not get an exact answer without storing the entire window**

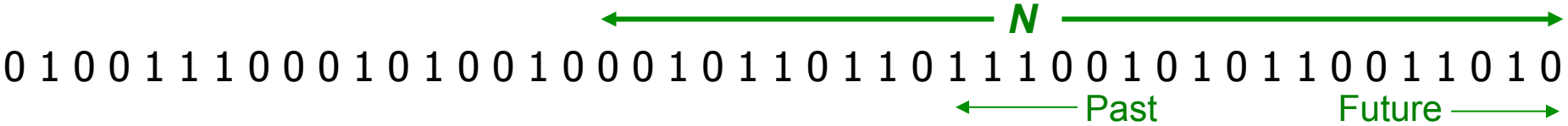
- **Real Problem:**
  What if we cannot afford to store N bits?
  - **E.g.**, we're processing 1 billion streams and N **= 1 billion**



- **But we are happy with an approximate answer**

An attempt: Simple solution

- **Q: How many 1s are in the last N bits?**
- A simple solution that does not really solve our problem: **Uniformity assumption**



- **Maintain 2 counters:**
  - S: number of 1s from the beginning of the stream
  - Z: number of 0s from the beginning of the stream
- **How many 1s are in the last N bits?**
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

DGIM Method

- **DGIM solution that does not assume uniformity**

- We store $O(\log^2 N)$ bits per stream

- **Solution gives approximate answer, never off by more than 50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

Summary

- **Sampling a fixed proportion of a stream**
  - Sample size grows as the stream grows
- **Sampling a fixed-size sample**
  - Reservoir sampling
- **Counting the number of 1s in the last N elements**
  - Exponentially increasing windows
  - Extensions:
    - Number of 1s in any last k (k < N) elements
    - Sums of integers in the last N elements

Beyond Naïve Bayes: Some Other Efficient [Streaming] Learning Methods

Shannon Quinn

(with thanks to William Cohen)

Rocchio's algorithm

- Relevance Feedback in Information Retrieval, SMART Retrieval System Experiments in Automatic Document Processing, 1971, Prentice Hall Inc.

Rocchio's algorithm

Many variants of these formulae

$$DF(w) = \#\,\text{different docs } w \text{ occurs in}$$

$$TF(w,d) = \#\,\text{different times } w \text{ occurs in doc } d$$

$$IDF(w) = \frac{|D|}{DF(w)}$$

$$u(w,d) = \log(TF(w,d)+1)\cdot\log(IDF(w))$$

…as long as $u(w,d)=0$ for words not in d!

$$\mathbf{u}(d) = \langle u(w_1,d),....,u(w_{|V|},d)\rangle$$

Store only non-zeros in $\mathbf{u}(d)$, so size is O(|d|)

$$\mathbf{u}(y) = \alpha\frac{1}{|C_y|}\sum_{d\in C_y}\frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2} - \beta\frac{1}{|D-C_y|}\sum_{d'\in D-C_y}\frac{\mathbf{u}(d')}{\|\mathbf{u}(d')\|_2}$$

But size of $\mathbf{u}(y)$ is O(|n_V|)

$$f(d) = \arg\max_y \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2}\cdot\frac{\mathbf{u}(y)}{\|\mathbf{u}(y)\|_2}$$

Rocchio's algorithm

Given a table mapping w to $DF(w)$, we can compute $\mathbf{v}(d)$ from the words in d… and the rest of the learning algorithm is just adding…

$$DF(w) = \#\text{ different docs } w \text{ occurs in}$$

$$TF(w,d) = \#\text{ different times } w \text{ occurs in doc } d$$

$$IDF(w) = \frac{|D|}{DF(w)}$$

$$u(w,d) = \log(TF(w,d)+1)\cdot \log(IDF(w))$$

$$\mathbf{u}(d) = \langle u(w_1,d),....,u(w_{|V|},d)\rangle, \quad \mathbf{v}(d) = \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2} = \langle v(w_1,d),....\rangle$$

$$\mathbf{u}(y) = \alpha \frac{1}{|C_y|}\sum_{d\in C_y}\mathbf{v}(d) - \beta\frac{1}{|D-C_y|}\sum_{d'\in D-C_y}\mathbf{v}(d), \quad \mathbf{v}(y) = \frac{\mathbf{u}(y)}{\|\mathbf{u}(y)\|_2}$$

$$f(d) = \text{argmax}_y\ \mathbf{v}(d)\cdot \mathbf{v}(y)$$

A hidden agenda

- Part of machine learning is good grasp of theory
- Part of ML is a good grasp of what hacks tend to work
- These are not always the same
  - Especially in big-data situations

- Catalog of useful tricks so far
  - Brute-force estimation of a joint distribution
  - Naive Bayes
  - Stream-and-sort, request-and-answer patterns
  - BLRT and KL-divergence (and when to use them)
  - **TF-IDF weighting – especially IDF**
    - it's often useful even when we don't understand why

Two fast algorithms

- Naïve Bayes: one pass
- Rocchio: two passes
  - if vocabulary fits in memory
- Both method are algorithmically similar
  - count and combine
- Thought thought thought thought thought thought thought thought thought thought thought experiment: what if we duplicated some features in our dataset many times times times times times times times times times times times?
  - e.g., Repeat all words that start with "t" "t" "t" "t" "t" "t" "t" "t" "t" "t" "t" ten ten ten ten ten ten ten ten ten ten times times times times times times times times times times times.
  - Result: some features will be **over-weighted** in classifier

This isn't silly – often there are features that are "noisy" duplicates, or important phrases of different length

Two fast algorithms

- Naïve Bayes: one pass
- Rocchio: two passes
  - if vocabulary fits in memory
- Both method are algorithmically similar
  - count and combine
- Result: some features will be **over-weighted** in classifier
  - unless you can somehow notice are correct for interactions/dependencies between features
- Claim: naïve Bayes is fast ***because*** it's naive

This isn't silly – often there are features that are "noisy" duplicates, or important phrases of different length