universidade de aveiro

Mining Large Scale Datasets

Mining Data Streams

(Adapted from CS246@Starford.edu; http://www.mmds.org)

Sérgio Matos - aleixomatos@ua.pt

Data Streams

- In many data mining situations, we do not know the entire dataset 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)

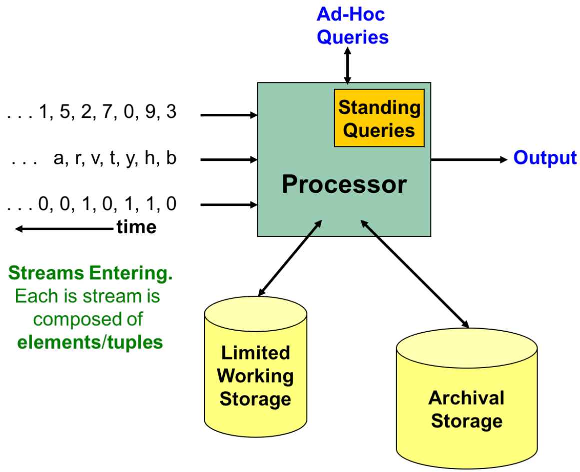
The Stream Model

- 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

  $\hookrightarrow$ How do you make critical calculations about the stream using a limited amount of (secondary) memory?

General Stream Processing Model



Applications

- Mining query streams
  - Google wants to know what queries are more frequent today than yesterday
- Mining click streams
  - Wikipedia 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., trending topics on Twitter, Facebook

Applications

- Sensor Networks
  - Many sensors feeding into a central controller
- Telephone call records
  - Data feeds into customer bills; settlements between telephone companies
- IP packets monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Different queries on Data Streams

- **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
- **Filtering a data stream**
  - Select elements with property x from the stream
- Counting distinct elements
  - Number of distinct elements in the last k elements
- Estimating moments
  - Estimate average or standard deviation of last k elements
- Finding frequent elements

Sampling from a Data Stream

- Since <u>we can not store the entire stream</u>, one obvious approach is to store a sample
- Two different problems
  1) Sample a fixed proportion of elements in the stream (maybe 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
     - Need to maintain the following property of the sample
       $\rightarrow$For all time steps k, each of k elements seen so far has equal probability of being sampled

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 day
  - 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

Q: 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)

Problem with Naïve Approach

Q: 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: $d/(x + d)$

Problem with Naïve Approach

Q: 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: 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
  - 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 $\frac{d/100}{x/10+d/100+18d/100} = \frac{d}{10x+19d}$

Solution: Sample users

- Pick $1/10^{th}$ of users and take all their searches in 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 the 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

Generalized solution

- Stream of tuples with keys
  - Key is some subset of the 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

To generate a 40% sample, hash into $b = 10$ buckets and take the tuple if it hashes to one of the first 4 buckets

Problem 2: Fixed-size sample

- Suppose we need to maintain a random sample S of size exactly s tuples
  E.g., limited main memory available

- 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 probability s/n
  - For example, $s = 2$
    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, 2/5
    At $n = 7$, each of the first 7 tuples is included in the sample S with equal probability, 2/7

Fixed size sample

Solution: **Reservoir Sampling**

- Store all the first s elements of the stream to S
- Suppose we have seen $n-1$ elements, and now the n$^{\text{th}}$ element arrives ($n > s$)
  - With probability s/n, keep the n$^{\text{th}}$ element, else discard it
  - If we picked the n$^{\text{th}}$ 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

Sliding windows

- Useful model of stream processing

  Queries are about a window of length N
  (the N most recent elements received)

- **Interesting case**

  N is so large that data cannot be stored in memory, or even on disk

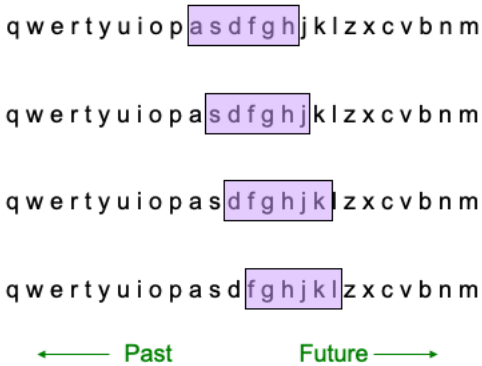
  Or, there are so many streams that windows for all cannot be stored

- Amazon example

  For every product X we keep 0/1 stream of whether that product was sold in the n^{th} transaction

  We want to query how many times we sold X in the last k sales

Sliding window: 1 stream



Counting bits

- Given a stream of 0s and 1s

  Be prepared to answer queries of the form

  How many 1s are in the last $k \leq N$ bits?

- Obvious solution

  Store the most recent N bits

  When new bit comes in, discard the $(N+1)^{th}$ bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 $\boxed{1\ 1\ 0\ 1\ 1\ 0}$

← Past Future →

Counting bits

- What if we cannot afford to store N bits?
  E.g., we are processing 1 billion streams and N is 1 billion

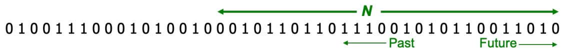
Counting bits

- What if we cannot afford to store N bits?
  E.g., we are processing 1 billion streams and N is 1 billion

- We can not get an exact answer without storing the entire window
- But an approximate answer may be sufficient

Counting bits: simple solution

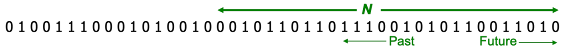
- How many 1s are in the last N bits?



- Maintain 2 counters
  - S: number of 1s from the beginning of the stream
  - Z: number of 0s from the beginning of the stream
- Number of 1s in the last N bits: $N \cdot \frac{S}{S+Z}$

Counting bits: simple solution

- How many 1s are in the last N bits?



- Maintain 2 counters

  S: number of 1s from the beginning of the stream
  Z: number of 0s from the beginning of the stream

- Number of 1s in the last N bits: $N \cdot \frac{S}{S+Z}$

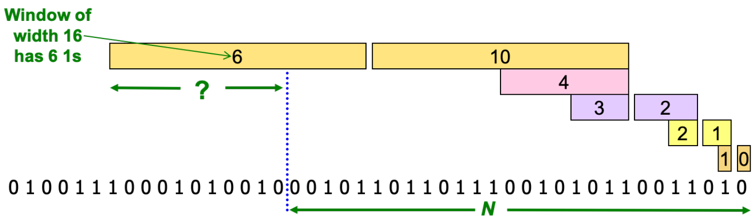
- Only works for uniform streams
  - What if distribution changes over time?

DGIM

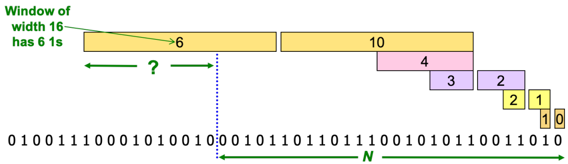
- Does not assume uniformity
- Store $O(log^2 N)$ bits per stream
- Gives approximate answer, never off by more than 50%
- Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more bits stored

DGIM: Exponential Windows

- Summarize exponentially increasing regions of the stream, looking backward
- Drop small regions if they begin at the same point as a larger region



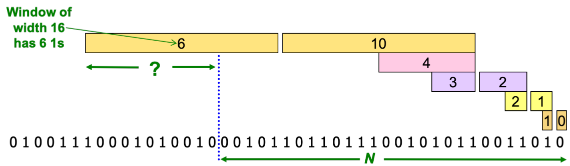
DGIM: Exponential Windows



Benefits

- Stores only $O(log^2 N)$ bits
  - $O(logN)$ counts of $log_2 N$ counts each
- Easy update as more bits enter
- Error in count no greater than the number of 1s in the "unknown" area

DGIM: Exponential Windows



Drawbacks

- As long as the 1s are fairly evenly distributed, the error due to the unknown region is small – no more than 50%
- But it could be that all the 1s are in the unknown area at the end

  In that case, the error is unbounded!

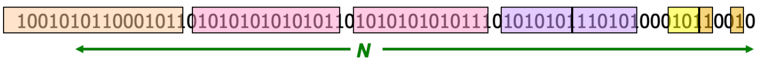
DGIM Method

- **Idea**

  Instead of summarizing fixed-length blocks, summarize blocks with specific number of 1s

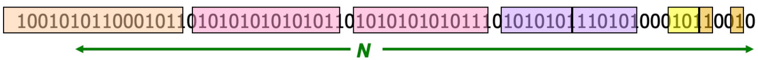
  Let the block sizes (number of 1s) increase exponentially

- When there are few 1s in the window, block sizes stay small, so errors are small

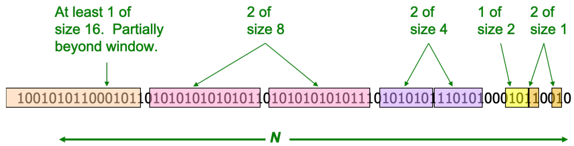


DGIM: Buckets

- Each bit in the stream has a timestamp
  - modulo N: 0, 1, ... N-1
  - Allows representing any relevant timestamp in $O(log_2 N)$ bits
- A bucket in DGIM is a record consisting of
  A) The timestamp of its end [O(log N) bits]
  B) Number of 1s between its start and end [O(log log N) bits]
- Constraint on buckets: Number of 1s must be a power of 2
  - That explains the O(log log N) in (B)



DGIM: Streams as Buckets

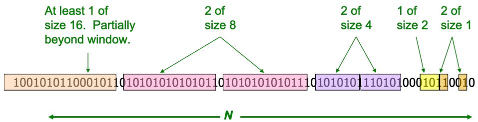


- Either one or two buckets with the same power-of-2 number of 1s
- Buckets do not overlap in timestamps
- Buckets are sorted by size

  Earlier buckets are not smaller than later buckets
- Buckets disappear when their end-time is > N time units in the past

Note: since we only consider a timestamp up to N (or -N), we can consider the current timestamp always as ‘0’, and keep a relative timestamp in the buckets. In this case, the relative timestamp needs to be updated (shifted) each time a new element comes through the stream

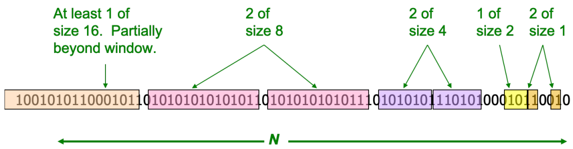
DGIM: Updating Buckets



- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to N time units before the current time
- If the current bit is 0: no other changes are needed
- If the current bit is 1
    1) Create a new bucket of size 1, for this bit, with end timestamp = current time
    2) If there are now three buckets of size 1, combine the oldest two into a bucket of size 2
    3) If there are now three buckets of size 2, combine the oldest two into a bucket of size 4
    4) And so on ...

DGIM: Updating Buckets

DGIM: Estimating counts



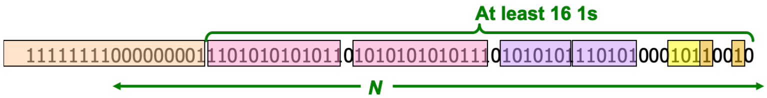
To estimate the number of 1s in the most recent N bits

- Sum the sizes of all buckets but the last
  bucket size = number of 1s in the bucket

- Add half the size of the last bucket

Remember: We do not know how many 1s of the last bucket are still within the wanted window

DGIM: Error bound



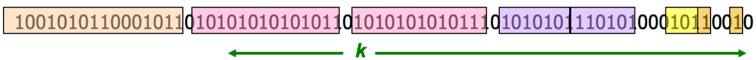
- Suppose the last bucket has size 2^r
- Then by assuming 2^{r-1} (i.e. half) of its 1s are still within the window, we make an error of at most 2^{r-1}
- Since there is at least one bucket of each of the sizes less than 2^r, the true sum is at least $1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1$
- Thus, the error is at most 50%

DGIM Extension: $k < N$

- Can also answer queries

  How many 1s in the last k, where $k < N$

- Find earliest bucket B that overlaps with k. Number of 1s is the sum of sizes of more recent buckets plus half the size of B

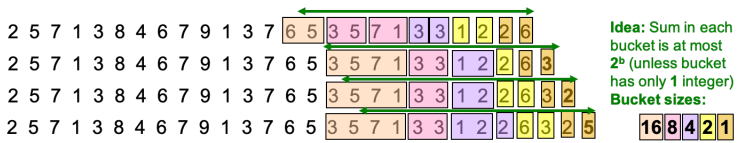


DGIM Extension: Stream of positive integers

We want the sum of the last k elements

1) If integers between 1 and 2^m, for some m
- Treat each of the m bits as a separate stream
- Use DGIM to estimate c_i = number of 1s in each (bit) stream
- The sum is $\sum_{i=0}^{m-1} c_i 2^i$

2) Use buckets to keep partial sums
   - Sum of elements in size b bucket is at most 2^b



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
  - We might be processing millions of filters on the same stream

Filtering Data Streams: Applications

- Example: Email spam filtering
  - We know 1 billion "good" email addresses
  - If an email comes from one of these, it is NOT spam

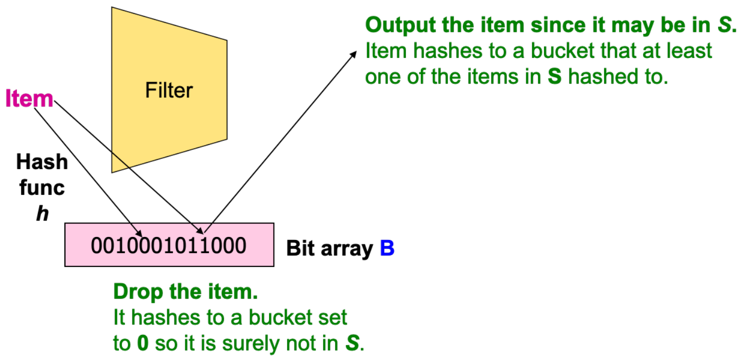
- Publish-subscribe systems
  - We are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user's interest

Filtering Data Streams: First solution

- Given a set of keys S that we want to filter
- Create a bit array B of n bits, initially all 0s
- Choose a hash function h with range $[0, n)$
- Hash each member of $s \in S$ to one of n buckets, and set that bit to 1, i.e., $B[h(s)] = 1$
- Hash each element a of the stream and output only those that hash to a bit that was set to 1

  Output a if $B[h(a)] == 1$

Filtering Data Streams: First solution



Creates false positives but no false negatives
If the item is in S we surely output it, if not we may still output it

Filtering Data Streams: First solution

$|S| = 1$ billion email addresses
$|B| = 1\text{GB} = 8$ billion bits

- If an email address is in S, then it surely hashes to a bucket that has the big set to 1, so it always gets through (no false negatives)
- Approximately $1/8$ of the bits are set to 1, so about $1/8^{\text{th}}$ of the addresses not in S get through to the output (false positives)
  - Actually, less than $1/8^{\text{th}}$, because more than one address might hash to the same bit

False positives

More accurate analysis for the number of false positives

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

False positives

We have m darts, n targets
What is the probability that a target gets at least one dart?

$$1 - (1 - 1/n)^m$$

False positives

We have m darts, n targets
What is the probability that a target gets at least one dart?

$$1 - (1 - 1/n)^m$$

$$= 1 - (1 - 1/n)^{n(m/n)}$$

$$\approx 1 - e^{-m/n}$$

False positives

We have m darts, n targets
What is the probability that a target gets at least one dart?

$$1 - e^{-m/n}$$

Fraction of 1s in B = probability of false positive = $1 - e^{-m/n}$

Example: 10^9 darts, $8 \cdot 10^9$ targets
Fraction of 1s in B = $1 - e^{-1/8} = 0.1175$
Compare with earlier estimate: $1/8 = 0.125$

Filtering Data Streams: Bloom filter

Consider: $|S| = m$, $|B| = n$

- Use k independent hash functions h_1, ..., h_k
- 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 $B[h_i(x)] = 1$ for all $i = 1, ..., k$ then declare that x is in S

  Otherwise discard the element x

Filtering Data Streams:Bloom filter

What fraction of the bit vector B are 1s?

- Throwing $k \cdot m$ darts at n targets
  So fraction of 1s is $1–e^{-km/n}$
- But we have k independent hash functions and we only accept element x if all k hash element x to a bucket of value 1
  So, false positive probability $= (1–e^{-km/n})^k$

Filtering Data Streams:Bloom filter

What fraction of the bit vector B are 1s?

- Throwing $k \cdot m$ darts at n targets
  So fraction of 1s is $1 – e^{-km/n}$

- But we have k independent hash functions and we only accept element x if all k hash element x to a bucket of value 1
  So, false positive probability $= (1 – e^{-km/n})^k$

Example
m = 1 billion, n = 8 billion
k = 1: $1 – e – 1/8 = 0.1175$
k = 2: $(1 – e – 1/4)^2 = 0.0493$

Bloom filter: summary

- 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
- Better to have 1 big B or k small Bs?
  It is the same: $(1-e^{-km/n})^k$ vs. $(1-e^{-m/(n/k)})^k$
  But keeping 1 big B is simpler

Different queries on Data Streams

- 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
- Filtering a data stream
  - Select elements with property x from the stream
- **Counting distinct elements**
  - Number of distinct elements in the last k elements
- **Estimating moments**
  - Estimate average or standard deviation of last k elements
- **Finding frequent elements**

Counting distinct elements

- Problem

  Data stream consists of a universe of elements chosen from a set of size N

  Maintain count of the number of distinct elements seen so far

- Obvious approach

  Maintain a hash table of all the distinct elements seen so far

Counting distinct elements

- Problem
  Data stream consists of a universe of elements chosen from a set of size N
  Maintain count of the number of distinct elements seen so far
- Obvious approach
  Maintain a hash table of all the distinct elements seen so far
- What if we do not have space to maintain the set of elements seen so far?
  - Estimate the count in an unbiased way
  - Accept that the count may have a little error, but limit the probability of large error

Applications

- How many different words are found among the Web pages being crawled at a site?

  Unusually low or high numbers could indicate artificial pages (spam?)

- How many different Web pages does each customer request in a week?

- How many distinct products have we sold in the last week?

Flajolet-Martin Approach

- Pick a hash function h that maps each of the N elements to at least $log_2 N$ bits

- For each stream element a, let $r(a)$ be the number of trailing 0s in $h(a)$

  $r(a) =$ position of first 1 counting from the right

  E.g., $h(a) = 12$, then 12 is 1100 in binary, so $r(a) = 2$

- Record R = the maximum $r(a)$ seen

  $R = max_a r(a)$, over all the items a seen so far

- **Estimated number of distinct elements =** 2^R

Flajolet-Martin Approach

Why Flajolet-Martin works?

- $h(a)$ hashes a with equal probability to any of N values
- Then $h(a)$ is a sequence of $log_2 N$ bits, where 2^{-r} fraction of all $h(a)$'s have a tail of r zeros
  - About 50% of as hash to ***0
  - About 25% of as hash to **00
- So, if longest tail saw is $R = 2$ (i.e., item hash ending ***100**) then we have probably seen about 4 distinct items so far
- So, it takes to hash about 2^r items before we see one with zero-suffix of length r

Flajolet-Martin Approach: In practice

- Use many hash functions h_i to get many estimates R_i
- Combine estimates

  Partition estimates into small groups

  Take the mean of each group

  Then take the median of the means as the final estimate

Moments

- Suppose a stream has elements chosen from a set A of N values
- Let m_i be the number of times value i occurs in the stream
- The k$^{\text{th}}$ moment is

$$\sum_{i \in A} (m_i)^k$$

- 0$^{\text{th}}$ moment = number of distinct elements
- 1$^{\text{st}}$ moment = count of the numbers of elements

  Length of the stream
- 2$^{\text{nd}}$ moment = surprise number S

  A measure of how uneven the distribution is

2^{nd} Moment: Surprise number

- Stream of length 100
- 11 distinct values
- Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9
  Surprise $S = 910$
- Item counts: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1
  Surprise $S = 8110$

AMS Method

- AMS method works for all moments
- Gives an unbiased estimate

- We will just concentrate on the 2$^{\text{nd}}$ moment S
- Pick and keep track of many variables X
  - For each variable X, store $X.el$ and $X.val$
    $X.el$ corresponds to item i
    $X.val$ corresponds to the count of item i
  - This requires a count in main memory, so number of Xs is limited

- Goal is to compute $S = \sum_i (m_i)^2$

AMS Method: Picking X

- Assume stream has length n
- Pick a random time t ($t < n$), so that any time is equally likely
- Set $X.el = i$, the item seen on the stream at time t
- Maintain count $X.val = c$ of the number of times item i occurs in the stream, starting from the chosen time t
- The estimate of the 2^{nd} moment is

$$S = f(X) = n(2 \cdot c - 1)$$

AMS Method: Picking X

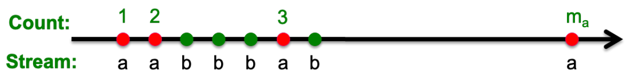
- Assume stream has length n
- Pick a random time t ($t < n$), so that any time is equally likely
- Set $X.el = i$, the item seen on the stream at time t
- Maintain count $X.val = c$ of the number of times item i occurs in the stream, starting from the chosen time t
- The estimate of the 2nd moment is

$$S = f(X) = n(2 \cdot c - 1)$$

- Actually, we keep track of multiple Xs (X_1, X_2,... X_k), and the final estimate is

$$S = \frac{1}{k} \sum_{j=1:k} f(X_j)$$

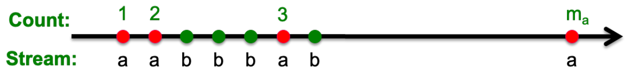
AMS Method: Analysis



- 2^{nd} moment is $S = \sum_i m_i^2$
- c_t = number of times item at time t appears from time t onwards

  $c_1 = m_a,\ c_2 = m_a - 1,\ c_3 = m_b$

AMS Method: Analysis

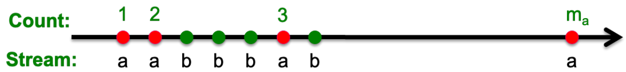


- 2^{nd} moment is $S = \sum_i m_i^2$
- c_t = number of times item at time t appears from time t onwards
  $c_1 = m_a,\ c_2 = m_a - 1,\ c_3 = m_b$

$$E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2 \cdot c_t - 1)$$

$$= \frac{1}{n}\sum_{i} n(1 + 3 + 5 + \ldots + 2m_i - 1)$$

AMS Method: Analysis



$$E[f(X)] = \frac{1}{n}\sum_i n(1+3+5+\ldots+2m_i-1)$$

$$(1+3+5+\ldots+2m_i-1) = \sum_{i=1}^{m_i}(2i-1) = 2\frac{m_i(m_i+1)}{2} - m_i = m_i^2$$

$$E[f(X)] = \frac{1}{n}\sum_i n(m_i)^2$$

$$E[f(X)] = \sum_i (m_i)^2 = S$$

Higher order moments

For estimating k[th] moment, just change the estimate

- For $k = 2$ we used $n(2c{-}1)$
- For $k = 3$ we use: $n(3c^2{-}3c + 1)$

Why?

- For $k = 2$, we had $(1 + 3 + 5 + \ldots + 2m_i - 1)$

  And showed [note that $2c - 1 = c^2 - (c - 1)^2$]

  $$\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c - 1)^2 = m^2$$

- For $k = 3$, $c^3 - (c - 1)^3 = 3c^2 - 3c + 1$

Generally: $n(c^k - (c - 1)^k)$

Estimating moments: In Practice

- Compute $f(X) = n(2c - 1)$
  for as many variables X as can fit in memory
- Average them in groups
- Take median of averages

- Problem: Streams never end
- We assumed there was a number n, the number of positions in the stream
- But real streams go on forever, so n is a variable – the number of inputs seen so far

Estimating moments: In Practice

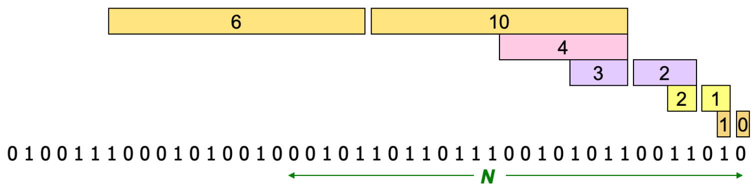
(1) The variables X have n as a factor
keep n separately; just hold the count in X

(2) Suppose we can only store k counts
We must throw some Xs out as time goes on

- Objective: Each starting time t is selected with probability k/n
- Solution: (fixed-size sampling)
  - Choose the first k times for k variables
  - When the n^{th} element arrives ($n > k$), choose it with probability k/n
  - If you choose it, throw one of the previously stored variables X out, with equal probability

Counting itemsets

- New Problem: Given a stream, which items appear more than s times in the window?
- Possible solution: Think of the stream of baskets as one binary stream per item
  - 1 = item present; 0 = not present
  - Use DGIM to estimate counts of 1s for all items



Counting itemsets

- In principle, we could count frequent pairs or even larger sets the same way
  - One stream per itemset
- Drawbacks
  - Only approximate
  - Number of itemsets is way too big

Exponentially Decaying Windows

A heuristic for selecting likely frequent item(sets)

- eg: what are "currently" most popular movies?
  - Instead of computing the raw count in last N elements
  - Compute a smooth aggregation over the whole stream
- If stream is $\mathbf{a_1}, \mathbf{a_2}, \ldots$ and we are taking the sum of the stream, take the answer at time $\mathbf{t}$ to be

$$\sum_{i=1}^{t} a_i (1-c)^{t-i}$$

  c is a small constant, maybe 10^{-6} or 10^{-9}

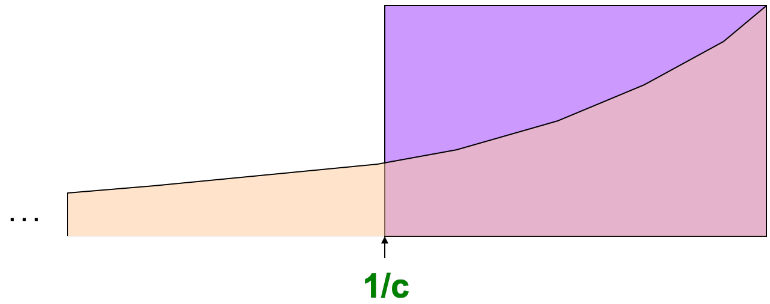
- When new $\mathbf{a_{t+1}}$ arrives
  Multiply current sum by $(\mathbf{1}-\mathbf{c})$ and add $\mathbf{a_{t+1}}$

Example: Counting Items

- If each a_i is an "item" we can compute the characteristic function of each possible item x as an Exponentially Decaying Window
- $\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$
  Where $\delta_i = 1$ if $a_i = x$, and 0 otherwise
- Imagine that for each item x we have a binary stream
  (1 if x appears, 0 if x does not appear)
- New item x arrives

  Multiply all counts by (1-c)
  Add +1 to count for element x
  Call this sum the "weight" of item x

Sliding Versus Decaying Windows



- Important property: sum over all weights $\sum_t (1-c)^t$ is $1/[1-(1-c)] = 1/c$

Example: Counting Items

- What are "currently" most popular movies?
- Suppose we want to find movies of weight $> 1/2$

- Important property: Sum over all weights
  $1/[1 - (1 - c)] = 1/c$

$\rightarrow$ No more than $2/c$ movies with weight of $1/2$ or more

$\hookrightarrow$ So, $2/c$ is a limit on the number of movies being counted at any time

Extension to Itemsets

- Count (some) itemsets in an Exponentially Decaying Window
  - What are currently "hot" itemsets
  - Too many itemsets to keep counts of all of them in memory

- When a basket B comes in
  - Multiply all counts by $(1 - c)$
  - For uncounted items in B, create new count
  - Add 1 to count of any item in B and to any itemset contained in B that is already being counted
  - Drop counts $< 1/2$
  - Initiate new counts

Initiation of New Counts

- Start a count for an itemset $S \subseteq B$ if every proper subset of S had a count prior to arrival of basket B
  - If all subsets of S are being counted this means they are "frequent/hot" and thus S has a potential to be "hot"

- Example

  Start counting $S = \{i, j\}$ iff both i and j were counted prior to seeing B

  Start counting $S = \{i, j, k\}$ iff $\{i, j\}$, $\{i, k\}$, and $\{j, k\}$ were all counted prior to seeing B

How many counts do we need?

- Counts for single items

  $< (2/c) \cdot$ (avg. number of items in a basket)

- Counts for larger itemsets $=$ **??**
- But we are conservative about starting counts of large sets
  - If we counted every set we saw, one basket of 20 items would initiate 1M counts