Mining Data Streams III / Frequent Itemsets II

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University
July 16, 2020

Today

Overview

- Mining Data Streams III
  - Estimating Moments: Alon-Matias-Szegedy algorithm
  - Decaying Windows
- Mining Frequent Itemsets II
  - The Multihash and Multistage Algorithms
  - Randomized Algorithms: Toivonen's Algorithm

Learning Goals: Understand these topics and get familiarized

Estimating Moments

The Alon-Matias-Szegedy Algorithm

Moments: Definition and Problem

Assume that the set of universal elements is ordered, and

- indexed by $1 \leq i \leq I$, where
- I is the cardinality of the universal set.

k-th Moment

Consider a stream.

- Let m_i be the number of occurrences of the i-th universal element in the stream
- The *k-th order moment of the stream* is defined to be

$$\sum_{i=1}^{I} (m_i)^k \tag{1}$$

Moments: Examples

$$\text{k-th order moment:} \quad \sum_{i=1}^{I} (m_i)^k$$

Examples

- The 0-th moment of a stream is the number of *distinct* stream elements
- The 1-st moment of a stream is the *overall* number of stream elements
- The 2-nd moment of a stream is sometimes called the *surprise number*
  - Consider a stream of length 100, on 11 different elements
  - The most even distribution, 10 appearances for one particular element, and 9 for all others, yields surprise number $10^2 + 10 \cdot 9^2 = 910$
  - The most uneven ("surprising") distribution, 90 appearances for one particular element, and 1 for all others, yields surprise number $90^2 + 10 \cdot 1^2 = 8110$

Alon-Matias-Szegedy Algorithm: Notation

- Keeping a count for each element in main memory is infeasible
- Therefore, we need to *estimate* the k-th order moments
  ☞ The *Alon-Matias-Szegedy algorithm* does this

Notation:

- Let n be the length of the stream
- Let X be variables for which we store attributes
  - $X.element$ as an element of the universal set
  - $X.index$ is a position $1 \leq i \leq n$ where $X.element$ appears
  - $X.value$ is defined as the number of times $X.element$ appears in the stream between (and including) positions $X.index$ and n

Alon-Matias-Szegedy Algorithm: Notation

Example

Let the stream be $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$.

- Stream length is $n = 15$
- The true second moment is $5^2 + 4^2 + 3^2 + 3^2 = 59$
- Let us keep three variables, X_1, X_2 and X_3, for which $X_1.index = 3, X_2.index = 8, X_3.index = 13$
- $X_1.element = c, X_2.element = d, X_3.element = a$
- $X_1.value = 3, X_2.value = 2, X_3.value = 2$

Along-Matias-Szegedy Algorithm: 2nd Moment

Alon-Matias-Szegedy Algorithm

- As an estimate for 2nd-order moment, compute, for any X,

$$n(2X.value - 1) \tag{2}$$

- For several X: 2nd-order moment = average of single estimates

Example (cont.): Stream = $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$

- We had $X_1.value = 3, X_2.value = 2, X_3.value = 2$
- $n(2X_1.value - 1) = 15(2 \cdot 3 - 1) = 75, n(2X_2.value - 1) = n(2X_3.value - 1) = 45$
- Yields average $(75 + 45 + 45)/3 = 55$, close to true value 59

Alon-Matias-Szegedy Algorithm: Proof I

- Let $e(i)$ be the stream element appearing at position i
- Let $c(i)$ be number of times $e(i)$ appears between (and including) positions i to n
- In example above, e.g. $e(6) = a$ and $c(6) = 4$

The expected value of $n(2X.value - 1)$ computes as the average of $n(2c(i) - 1)$ over i:

$$E(n(2X.value - 1)) = \frac{1}{n} \sum_{i=1}^{n} n(2c(i) - 1) \tag{3}$$

by canceling factors further simplifying to

$$\sum_{i=1}^{n} (2c(i) - 1) \tag{4}$$

Alon-Matias-Szegedy Algorithm: Proof II

$$E(n(2X.value - 1)) = \sum_{i=1}^{n}(2c(i) - 1) \tag{5}$$

Regroup summands in (5) by their associated values $e(i)$:

$$\sum_{i=1}^{n}(2c(i) - 1) = \sum_{a} \sum_{i:\ e(i)=a} (2c(i) - 1) \tag{6}$$

Consider one particular a, let m_a be the number of times a appears in stream:

- Last i where a appears: $2c(i) - 1 = 2 \times 1 - 1 = 1$
- Second last i where a appears: $2c(i) - 1 = 2 \times 2 - 1 = 3$
- $\vdots$
- First i where a appears: $2 \times m_a - 1$

Alon-Matias-Szegedy Algorithm: Proof III

Consider one particular a, let m_a be the number of times a appears in stream:

- Last i where a appears: $2c(i) - 1 = 2 \times 1 - 1 = 1$
- Second last i where a appears: $2c(i) - 1 = 2 \times 2 - 1 = 3$
- $\vdots$
- First i where a appears: $2 \times m_a - 1$

This yields

$$\sum_{i:e(i)=a} (2c(i) - 1) = 1 + 3 + 5 + ... + (2m_a - 1) = (m_a)^2 \tag{7}$$

where the last equation follows from an easy induction.

Alon-Matias-Szegedy Algorithm: Proof IV

This yields

$$\sum_{i:e(i)=a} (2c(i) - 1) = 1 + 3 + 5 + ... + (2m_a - 1) = (m_a)^2$$

where the last equation follows from an easy induction.

Overall,

$$E(n(2X.value - 1)) \overset{(5)}{=} \sum_{i=1}^{n} (2c(i) - 1) \overset{(6)}{=} \sum_{a} \sum_{i:\ e(i)=a} (2c(i) - 1) \overset{(7)}{=} \sum_{a} (m_a)^2 \quad (8)$$

which concludes the proof. □

Estimating Higher-Order Moments

- Observe that $2v - 1$ for $v = 1, \dots, m_a$ sum to $(m_a)^2$
- The inductive proof makes use of the "telescope property": $2v - 1 = v^2 - (v-1)^2$
- Analogously, by $v^3 - (v-1)^3 = 3v^2 - 3v + 1$:

$$\sum_{v=1}^{m_a} 3v^2 - 3v + 1 = (m_a)^3 \tag{9}$$

- So, for a variable X, we can use

$$n(3((X.value)^2 - 3X.value + 1)) \tag{10}$$

  as an estimate for the third order moment
- For arbitrary k, take

$$n((X.value)^k - (X.value - 1)^k) \tag{11}$$

  as estimate for variable X

MOMENTS FOR INFINITE STREAMS

- *Situation:* Stream length n grows with time
- *Problem:* For selected variables X, we need $X.index$ to be uniformly distributed
- So, selecting variables X a priori tends to be biased, ☞ non-uniform
- *Solution:* Maintain as many variables as possible. As stream grows:
  - Discard existing variables
  - Replace by new ones
  - such that at all times, variables are uniformly distributed
- *Remark:* This establishes a generally applicable strategy for sampling elements from a stream:
  - Recall the problem of selecting representative samples
  - Recall the general sampling problem

Moments for Infinite Streams: Solution

- Suppose we can store/maintain s variables
- Suppose we have seen n stream elements
- Suppose the s different $X.index$ are uniformly distributed
- That is, the probability to see position $1 \leq i \leq n$ among the selected $X.index$ is s/n
- Upon arrival of $(n+1)$-st element, do
  - Pick position $n+1$ with probability $s/(n+1)$
  - If picked, create variable X with $X.index = n+1$, and throw out any earlier X with equal probability $1/s$
  - If not picked, keep existing variables
- *Claim:* Afterwards, each position has been selected with probability $s/(n+1)$

Moments for Infinite Streams: Solution

- Upon arrival of $(n+1)$-st element, do
  - Pick position $n+1$ with probability $s/(n+1)$
  - If picked, create variable X with $X.index = n+1$, and throw out any earlier X with equal probability $1/s$
  - If not picked, keep existing variables
- *Claim:* Afterwards, each position has been selected with probability $s/(n+1)$

Proof:

- $(n+1)$-st position is picked with probability $s/(n+1)$
- Let $1 \leq i \leq n$ any other position: proof by induction
- Induction hypothesis: before $(n+1)$-st element arrived, i had been picked with probability s/n
- With probability $1 - s/(n+1)$, probability for having i stays s/n
- With probability $s/(n+1)$, probability for having i is $(s-1)/s$

Moments for Infinite Streams: Solution

Proof:

- $(n+1)$-st position is picked with probability $s/(n+1)$
- With probability $1 - s/(n+1)$, probability for having i stays s/n
- With probability $s/(n+1)$, probability for having i is $(s-1)/s$

Overall

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s}{n+1})(\frac{s-1}{s})(\frac{s}{n}) \tag{12}$$

simplifying to

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s-1}{n+1})(\frac{s}{n}) = ((1 - \frac{s}{n+1}) + (\frac{s-1}{n+1}))(\frac{s}{n}) \tag{13}$$

yielding

$$(\frac{n}{n+1})(\frac{s}{n}) = \frac{s}{n+1} \tag{14}$$

□

Most Common Elements

Decaying Windows

Decaying Windows: Motivation

- *Stream:* Movie tickets purchased all over the world
- *Goal:* Summarize stream by listing currently most "popular" movies
- *Currently popular:*
  - Movie that sold plenty of tickets years ago not to be listed
  - Movie that sold $2n$ tickets last week, for large n, currently popular
  - Movie that sold n tickets in last 10 weeks is even more popular
  - How to grasp that idea?

Decaying Windows: Motivation

- *Stream:* Movie tickets purchased all over the world
- *Goal:* Summarize stream by listing currently most "popular" movies
- *Possible solution:*
  - Bit stream for each movie
  - The i-th bit in a movie stream is 1 if the i-th ticket was for that movie
  - Pick window of size N, where N reasonably chosen to reflect tickets to be recent
  - Use method for estimating number of ones to estimate number of tickets for each movie
  - Rank movies by the estimated counts
  - Works for movies, because there only thousands of movies
  - *Drawback:* Does not work for items at Amazon or tweets per Twitter-user ☞ too many items or users

Decaying Windows: Motivation

- *Stream:* Movie tickets purchased all over the world
- *Goal:* Summarize stream by listing currently most "popular" movies
- *Alternative approach:*
  - Do not ask for count of ones in a window
  - Rather, compute "smooth aggregation" of all ones in stream
  - Smooth: use weights to rate stream elements in terms of recentness
  - The further back in the stream, the less weight given

Exponentially Decaying Window: Definition

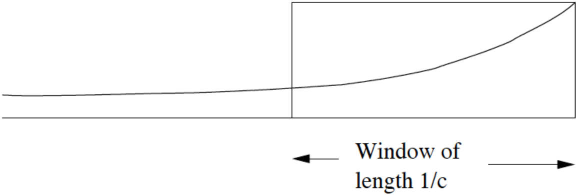
Definition [Exponentially Decaying Window]:

Let a stream

- consist of elements $a_1, a_2, \ldots, a_t$ (where a_t is the most recent one)
- Let c be small constant, e.g. $c \in [10^{-9}, 10^{-6}]$

The *exponentially decaying window* for the stream is defined to be the sum

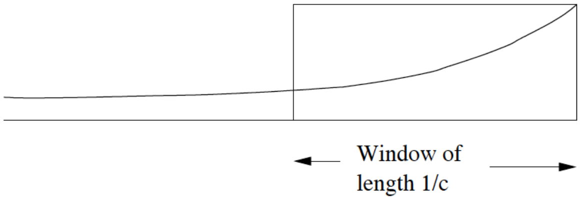
$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i \quad (15)$$



Decaying window and fixed-length window of equal weight

From mmds.org

Exponentially Decaying Window: Definition



Decaying window and fixed-length window of equal weight

From `mmds.org`

- Decaying window puts weight $(1 - c)^i$ on $(t - i)$-th element
- A window of length $1/c$ puts equal weight 1 on the first $1/c$ elements
- Both principles distribute the same weight to stream elements overall

Updating Exponentially Decaying Windows

Upon arrival of a new element a_{t+1}, one updates the exponentially decaying window $\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$ by

1. multiplying the current window by $(1-c)$, yielding

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^{i+1}$$

2. adding a_{t+1}, yielding

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^{i+1} + a_{t+1} = \sum_{i=0}^{(t+1)-1} a_{(t+1)-i}(1-c)^i$$

Exponentially Decaying Windows: Finding the Most Popular Movies

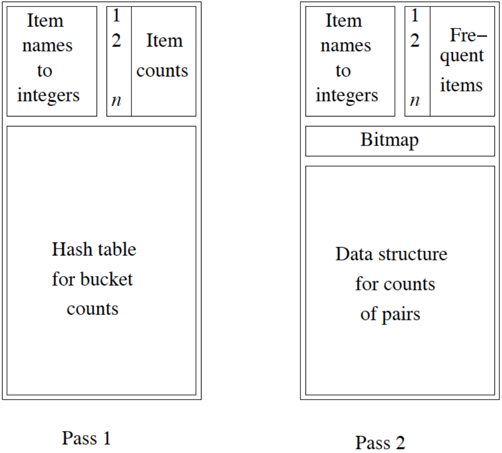
- *Most Popular Movies: Idea*
  - Have a bit stream for each movie, as before
  - Use e.g. $c = 10^{-9}$ ($\approx$ sliding window of size $1/c = 10^9$)
  - On incoming movie ticket sale, update all decaying windows, as described above
    - First, multiply all decaying windows by $1 - c$
    - Add 1 for stream of the movie of the ticket; if there is no stream for that movie, create one
    - Do nothing (add 0) for all other streams
  - If any decaying window drops below threshold of $1/2$, drop window
  - Because the sum of all scores is $1/c$, there cannot be more than $2/c$ movies with score of $1/2$ or more
  - So, $2/c$ is limit on number of movies being tracked at any time
  - In practice, there should be much less movies counted
- *Therefore,* one can apply the technique also for Amazon items and Twitter-users

A-Priori Algorithm Extensions
The Multistage Algorithm

The Multistage Algorithm: Motivation

- The predominant bottleneck in most applications of A-Priori is the size of C_2, the candidate pairs
- Several algorithms address to trim down that size
- Exemplary algorithms:
  - The algorithm of Park, Chen and Yu *(PCY algorithm)*
  - The Multistage algorithm
  - The Multihash algorithm
- Treated PCY before; will do Multistage and Multihash in the following

Multistage Algorithm: Main Memory Usage



Use of main memory during PCY passes

Adopted from `mmds.org`

The Multistage Algorithm

- *Particular Motivation:* Selecting $\{i, j\}$ to be in C_2
- In PCY: even when reducing to frequent i and j, and $\{i, j\}$ hashing to frequent buckets, still too many pairs to be counted
- So, need to decrease size of C_2 further
- Do this by introducing extra pass:
  - The *first pass* is as before in PCY
  - In the *second pass*, have another hash table that raises a third condition
  - In the *third pass*, count only pairs that fulfill all three conditions

The Multistage Algorithm: Second Pass

- Maintain,
  - tables A on item names to integers and
  - C on frequent items: $C[i] = k$ if item i is k-th frequent item, and $C[i] = 0$ if i-th item is not frequent
  - bitmap H', where entries refer to buckets from hash map, with 1 indicating frequent bucket
- In addition, raise another hash table H_2 that
  - hashes pairs of items $\{i, j\}$
  - if both i and j are frequent (*), and $\{i, j\}$ hashes to bucket b such that $H'[b] = 1$ (**), to
  - buckets holding integers

$$H_2[\{i, j\}] \in \mathbb{N}$$

    reflecting number of times pairs hashed to that bucket

The Multistage Algorithm: Second Pass

- To construct H_2, use double loop through baskets:
  - hash each pair that meets (*) and (**) to bucket, and
  - increase the integer in that bucket by one
- Again, a *frequent bucket b* in H_2 exceeds the support threshold s
- Relative to number of frequent buckets using first H, the number of frequent buckets in H_2 should be much reduced, because much less pairs are hashed

The Multistage Algorithm

- *Definition of Multistage C_2:* For $\{i,j\} \in C_2$, both
  - (*) i and j must be frequent
  - (**) $\{i,j\}$ must hash to a frequent bucket according to H
  - (***) $\{i,j\}$ must hash to a frequent bucket according to H_2
- *Use of C_2 in third pass:*
  - Keep A (items to integers), C (frequent items), H' (bitmap for H)
  - Transform H_2 into bitmap H'' where

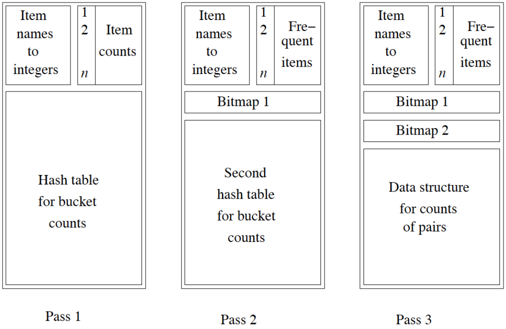
$$H''[b] = \begin{cases} 1 & \text{if } H_2[\{i,j\}] \geq s \\ 0 & \text{if } H_2[\{i,j\}] < s \end{cases} \tag{16}$$

where b is the bucket $\{i,j\}$ hashes to by H_2

The Multistage Algorithm

- *(Tricky?) Question:* Why does (***) not imply (**) and (*)? Weren't all $\{i, j\}$ hashed with H_2 selected to hash to frequent bucket with H and consist of frequent i and j?
- *Answer:*
  - *Yes:* for the second part.
  - *But:* Any $\{i, j\}$ that does not consist of frequent i, j, or hash to frequent bucket with H could hash to frequent bucket with H_2 nevertheless, although not having contributed to count in the bucket it hashes to

MULTISTAGE ALGORITHM: MAIN MEMORY USAGE



Use of main memory during Multistage passes

Adopted from `mmds.org`

A-Priori Algorithm Extensions
The Multihash Algorithm

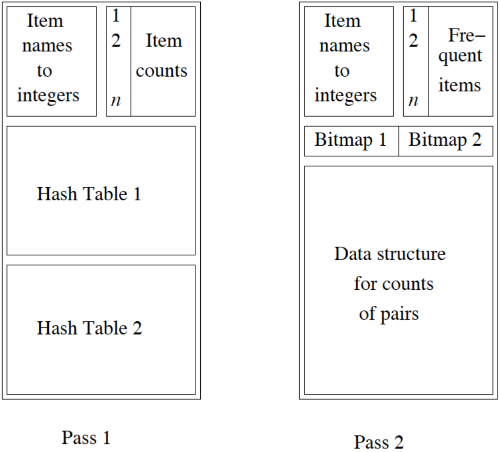
The Multihash Algorithm

- *Particular Motivation:* Try to profit from virtues of Multistage algorithm in one, and not two passes
- So, in *first pass*, use two hash tables H_1 and H_2,
- Both H_1 and H_2 have only half as many buckets
- *Applicability:* When average bucket size in PCY is much lower than threshold s
  ☞ So, still number of frequent buckets will be limited when using half as many buckets
- For proceeding with second pass, turn H_1 and H_2 into bitmaps H', H'' as in Multistage
- Apply exact same conditions as in Multistage for pair $\{i, j\}$ to be counted

The Multihash Algorithm: Example

- Imagine average bucket count in PCY is $s/10$
- Number of pairs of items randomly hashing to frequent bucket is $1/10$
- So, with half as many buckets, average count in Multihash is $s/5$
- Number of pairs of items randomly hashing to frequent buckets with both H_1 and H_2 is $1/25$
- So, we deal with (approximately!) 2.5 times less frequent pairs in Multihash

MULTIHASH ALGORITHM: MAIN MEMORY USAGE



Use of main memory during A-Priori passes

Adopted from `mmds.org`

Limited-Pass Algorithms
The Toivonen Algorithm

Limited-Pass Algorithms

Strategy

- To save on main memory, consider only a subsample of baskets
- Take into account that one may have
  - *False negatives:* itemsets not identified as frequent although they are
  - *False positives:* itemsets identified as frequent although they are not
- In many applications, a certain amount of false negatives and/or positives is acceptable

Algorithms

- *Simple Randomized Algorithm:* basic strategy is briefly discussed
- *Savasere, Omiecinski, Navate (SON):* not considered in the following
- *Toivonen:* explained here

Simple Randomized Algorithm: Strategy

- Let m be the overall number of baskets
- Consider a situation where main memory can deal with only k baskets
- Select probability p such that $pm = k$
- Run through basket file, and select each basket to be part of sample with probability p
- If s is original support threshold, set $s' := sp$ for sample
- Run any A-Priori type algorithm on resulting subset of baskets using s' as support threshold
- Declare itemsets frequent in subsample as frequent overall

Simple Randomized Algorithm: Errors

- *False positive:* Itemset that is frequent in sample, but not in the whole
- *False negative:* Itemset that is frequent in the whole, but not in sample
- *Eliminating false positives:* Running through whole dataset and counting each itemset found to be frequent in the sample eliminates false positives entirely
- *Eliminating false negatives:* Cannot eliminate false negatives entirely, but reduce them by choosing $s' < sp$, e.g. $s' = 0.9sp$

TOIVONEN'S ALGORITHM I

Algorithm

- Run simple sample strategy at $s' = 0.9ps$ or $s' = 0.8ps$
- Constructs all itemsets that are frequent in the sample (at support threshold s')
- Subsequently, construct *negative border*, which contains
  - itemsets that are not frequent in the sample
  - while all of their immediate subsets are frequent in the sample

Negative Border: Example

- Consider items $\{A, B, C, D, E\}$
- Itemsets found to be frequent: $\{A\}, \{B\}, \{C\}, \{D\}, \{B, C\}, \{C, D\}$, formally also the empty set $\emptyset$
- Negative border:
  - $\{E\}$ not frequent, but $\emptyset$ is frequent
  - $\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}$: not frequent, but singletons contained are
  - No triples in negative border ($\{B, C, D\}$ is not, because $\{B, D\}$ is not frequent)

Toivonen's Algorithm II

- *Pass through full dataset:* Count all itemsets found to be frequent or in the negative border in the sample
- Two possible outcomes:
    1. No member of negative border is frequent in whole dataset: correct set of itemsets frequent in the whole are the ones frequent in the sample found to be frequent in the whole
    2. Some member of negative border is frequent in whole dataset: there could be even larger sets frequent in the whole
       ☞ no guarantees, repeat the algorithm

Toivonen's Algorithm: Proof

- *No false positives:* all frequent itemsets were determined as frequent in the whole dataset ✓
- *No false negatives:* If no member of the negative border is frequent in the whole dataset, we need to show that there is no itemset that
  - is frequent in the whole
  - while, in the sample not among the frequent itemsets
  - while, in the sample, not in the negative border

Toivonen's Algorithm: Proof

- *Proof of no false negatives:* Suppose the contrary, that is, there is S found to be frequent in the whole, but is neither frequent in the sample nor part of the negative border of the sample
- By monotonicity, all subsets of S are frequent in the whole
- Choose $T \subseteq S$ of the smallest possible size such that still T is not frequent in the sample
- *Claim:* T is in the negative border of the sample
- *Proof of Claim:*
  - All proper subsets of T are frequent in the sample, because T was chosen of the smallest possible size
  - T itself is not frequent in the sample
- We obtain that T was in the negative border of the sample, but frequent in the whole, which is a contradiction!

Materials / Outlook

- See *Mining of Massive Datasets*, chapter 4.5, 4.7; 6.3.2, 6.3.3, 6.4.1, 6.4.2, 6.4.5, 6.4.6
- As usual, see `http://www.mmds.org/` in general for further resources