Algorithms for Data Science
Course Unit 4

Mining (Large) Data Streams

Spyros Kontogiannis & Christos Zaroliagis

Slides based on those in: http://www.mmds.org
New Topic: Infinite Data

High dim. data
- Locality sensitive hashing
- Clustering
- Dimensionality reduction

Graph data
- PageRank, SimRank
- Community Detection
- Spam Detection

Infinite data
- Filtering data streams
- Queries on streams
- Web advertising

Machine learning
- SVM
- Decision Trees
- Perceptron, kNN

Apps
- Recommender systems
- Association Rules
- Duplicate document detection
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?
Stochastic Gradient Descent (SGD) is an example of a stream algorithm [cf. CH12 – Large-Scale ML]

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:  
  1. First train the classifier on training data.
  2. For every example from the stream, we slightly update the model (using small learning rate)
General Stream Processing Model

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

... 1, 5, 2, 7, 0, 9, 3
... a, r, v, t, y, h, b
... 0, 0, 1, 0, 1, 1, 0
time

Processor

Ad-Hoc Queries

Standing Queries

Output

Limited Working Storage

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

- **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
Types of queries one wants on answer on a data 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: Sampling a fixed proportion

As the stream grows the sample also gets bigger
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 MUST HAVE equal probability of being sampled
Problem 1: Sampling fixed proportion

Scenario: Search engine query stream
- Stream of tuples: (user or IP, query, time)

- Answer questions such as: How often did a typical user run the same query in a single day...
- Have space to store $1/10$th 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
    $$\frac{\frac{d}{100}}{x + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10x + 19d}$$
Solution: Sample Users

Solution:

- Pick \( \frac{1}{10} \)th of users and take all their searches in the sample.

- Do not take any search for unpicked users.

**QUESTION:** How do we sample users?

**ANSWER:** Use a (random) hash function that uniquely hashes the user name or user ID uniformly into 10 buckets.
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 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: $[a \; x \; c \; y \; z \; k \; c \; d \; e \; g \ldots$

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: Store all the $n$ tuples seen so far and out of them pick $s$ at random
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

- 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:

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

- 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 \( n+1 = \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1} \)
Queries over a (long) Sliding Window
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 on a single stream:

\[
\begin{align*}
q w e r t y u i o p a & \quad s d f g h j k l z x c v b n m \\
q w e r t y u i o p a & \quad s d f g h j k l z x c v b n m \\
q w e r t y u i o p a & \quad s d f g h j k l z x c v b n m \\
q w e r t y u i o p a & \quad s d f g h j k l z x c v b n m \\
q w e r t y u i o p a & \quad s d f g h j k l z x c v b n m \\
\end{align*}
\]

\[\text{Past} \quad \text{Future}\]
Problem:

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

Obvious solution:

Store the most recent $N$ bits

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

Suppose $N=6$
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?** $N \cdot \frac{S}{S+Z}$
- **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
Solution that doesn’t (quite) work:

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

We can reconstruct the count of the last $N$ bits, except we are not sure how many of the last 6 1s are included in the $N$. 

Window of width 16 has 6 1s
What’s Good?

- Stores only $O(\log^2 N)$ bits
  - $O(\log N)$ counts of $\log_2 N$ bits each
- Easy update as more bits enter
- Error in count no greater than the number of 1s in the “unknown” area
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!
Fixup: 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: Timestamps

- Each bit in the stream has a timestamp, starting 1, 2, ...
- Record timestamps modulo $N$ (the window size), so we can represent any relevant timestamp in $O(\log_2 N)$ bits
A **bucket** in the DGIM method is a record consisting of:

- **(A)** The timestamp of its end \( [O(\log N) \text{ bits}] \)
- **(B)** The number of 1s between its beginning and end \( [O(\log \log N) \text{ bits}] \)

**Constraint on buckets:**
Number of 1s must be a power of 2
- That explains the \( O(\log \log N) \) in **(B)** above
Either **one** or **two** buckets with the same size (size = 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 **smaller than** $n - N$ (i.e., more than $N$ time units in the past)
Three properties of buckets that are maintained:
- 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
Updating Buckets (1)

- 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.

- **2 cases:** Current bit is 0 or 1

- **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 just this bit
   - 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 …
Example: Updating Buckets

Current state of the stream:

01010110001011010101010101010110101010101011101010111000101100101

Bit of value 1 arrives

00101011000010110101010101010110101010101011101010101110101000101100101

Two orange buckets get merged into a yellow bucket

001010110001011010101010101011010101010101110101010111010100010110010110101010101010110101010101011101010101110101000101100101

Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:

010110001011010101010101011010101010101110101010111010100010110010110101010101010110101010101011101010101110101000101100101

Buckets get merged…

010110001011010101010101011010101010101110101010111010100010110010110101010101010110101010101011101010101110101000101100101

State of the buckets after merging

010110001011010101010101011010101010101110101010111010100010110010110101010101010110101010101011101010101110101000101100101
To estimate the number of 1s in the most recent $N$ bits:

1. Sum the sizes of all buckets but the last
   (note “size” means the number of 1s in the bucket)
2. 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
Example: Bucketized Stream

At least 1 of size 16. Partially beyond window.

2 of size 8

2 of size 4
1 of size 2
2 of size 1

N
Why is error 50%? Let’s prove it!

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 $1 + 2 + 4 + \ldots + 2^{r-1} = 2^r - 1$

Thus, error at most 50%
Further Reducing the Error

- Instead of maintaining 1 or 2 of each size bucket, we allow either \( r-1 \) or \( r \) buckets (\( r > 2 \))
  - Except for the largest size buckets; we can have any number between 1 and \( r \) of those
- Error is at most \( O(1/r) \)
- By picking \( r \) appropriately, we can tradeoff between number of bits we store and the error
Can we use the same trick to answer queries 

**How many 1’s in the last \( k \)?** where \( k < N \)?

- **A:** Find earliest bucket \( B \) that at overlaps with \( k \).
  Number of **1s** is the **sum of sizes of more recent buckets** + \( \frac{1}{2} \) size of \( B \)

---

Can we handle the case where the stream is not bits, but integers, and we want the sum of the last \( k \) elements?
Stream of positive integers
We want the sum of the last \( k \) elements

- **Amazon:** Avg. price of last \( k \) sales

**Solution:**

1. If you know all have at most \( m \) bits
   - Treat \( m \) bits of each integer as a separate stream
   - Use DGIM to count 1s in each integer \( c_i \) ... estimated count for \( i \)-th bit
   - 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 \( 2^b \)

---

**Idea:** Sum in each bucket is at most \( 2^b \) (unless bucket has only 1 integer)

**Bucket sizes:**
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
More Algorithms for Streams...

- **(1) Filtering a data stream:** Bloom filters
  - Select elements with property $x$ from stream

- **(2) Counting distinct elements:** Flajolet-Martin
  - Number of distinct elements in the last $k$ elements of the stream

- **(3) Estimating moments:** AMS method
  - Estimate std. dev. of last $k$ elements

- **(4) Counting frequent items**
(1) 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
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 **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 bit that was set to **1**
  - Output $a$ only if $B[h(a)] = 1$
First Cut Solution (2)

- Creates false positives but no false negatives
  - if the item IS in \( S \), then we surely output it
  - if the item IS NOT in \( S \), then we may still output it

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.

Drop the item. It hashes to a bucket set to 0 so it is surely not in \( S \).
First Cut Solution (3)

- $|S| = 1$ billion email addresses
- $|B| = 1$GB $= 8$ billion bits
- if the email address IS in $S$, then it hashes to a bucket whose bit is set to 1 (so it always gets through $\Rightarrow$ no false negatives)
- Approximately $1/8$ of the bits are set to 1, so about $1/8^{th}$ of the addresses not in $S$ get through to the output (false positives)
  - Actually, less than $1/8^{th}$, because more than one address might hash to the same bit
Analysis: Throwing Darts (1)

- 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
We have \( m \) darts, \( n \) targets

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

\[
1 - \frac{1}{e^{m/n}}
\]
Fraction of 1s in the array $B$ = probability of false positive = $1 - e^{-m/n}$

Example: $m = 10^9$ darts, $n = 8 \cdot 10^9$ targets...

- Fraction of 1s in $B = 1 - e^{-1/8} = 0.1175$
- Compare with earlier (rough) estimate: $1/8 = 0.125$
Consider: $|S| = m$, $|B| = n$

- Use $k$ independent hash functions $h_1, \ldots, h_k$

**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 $i = 1, \ldots, k$)

**Run-time:**
- When a stream element with key $x$ arrives
  - IF $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$
    THEN declare that $x$ is in $S$
    ... $x$ hashes to a bucket set to 1 for every hash function $h_i(x)$ ...
  - ELSE discard the element $x$

*(note: we have a single array $B!$)*
Bloom Filter – Analysis (1)

- 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 let the element \( x \) through if all \( k \) hash element \( x \) to a bucket of value 1

- So, false positive probability = \( (1 - e^{-km/n})^k \)
Bloom Filter – Analysis (2)

- $m = 1$ 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 $k = 8 \ln(2) = 5.54 \approx 6$
    - Error at $k = 6$: $(1 - e^{-1/6})^2 = 0.0235$
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

- Is it 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
(2) Counting Distinct Elements
Problem:
- Data stream consists of a universe of elements chosen from a set of size $N$
- Maintain a count of the number of distinct elements seen so far

Obvious approach:
Maintain the set of elements seen so far
- That is, keep a hash table of all the distinct elements seen so far
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?
Real problem: 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 that the error is large
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 $h(a)$
  - $r(a) = \text{position of first 1 counting from the right}$
    - E.g., say $h(a) = 12$, then $12$ is $1100$ in binary, so $r(a) = 2$
  - Record $R = \text{the maximum } r(a) \text{ seen}$
    - $R = \max_a r(a)$, over all the items $a$ seen so far

- Estimated number of distinct elements $= 2^R$
Why It Works: Intuition

- **Very very rough and heuristic intuition why Flajolet-Martin works:**
  - $h(a)$ hashes $a$ with equal prob. to any of $N$ values
  
  - Then $h(a)$ is a sequence of $\log_2 N$ bits, where $2^{-r}$ fraction of all $a$s have a tail of $r$ zeros
    - About 50% of $a$s hash to ***0
    - About 25% of $a$s hash to **00
    - So, if we saw the longest tail of $r=2$ (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 $r$**
Now we show why Flajolet-Martin works

Formally, we will show that probability of finding a tail of $r$ zeros:

- Goes to 1 if $m \gg 2^r$
- Goes to 0 if $m \ll 2^r$

where $m$ is the number of distinct elements seen so far in the stream

Thus, $2^R$ will almost always be around $m!$
Why It Works: More formally

- What is the probability that a given $h(a)$ ends in at least $r$ zeros is $2^{-r}$
  - $h(a)$ hashes elements uniformly at random
  - Probability that a random number ends in at least $r$ zeros is $2^{-r}$
- Then, the probability of NOT seeing a tail of length $r$ among $m$ elements:
  \[
  (1 - 2^{-r})^m
  \]
  - Prob. all end in fewer than $r$ zeros.
  - Prob. that given $h(a)$ ends in fewer than $r$ zeros
Why It Works: More formally

- Note: \((1 - 2^{-r})^m = (1 - 2^{-r})^{2^{r}(m2^{-r})} \approx e^{-m2^{-r}}\)

- Prob. of NOT finding a tail of length \(r\) is:
  - If \(m \ll 2^r\), then prob. tends to 1
    - \((1 - 2^{-r})^m \approx e^{-m2^{-r}} = 1\) as \(m/2^r \to 0\)
    - So, the probability of finding a tail of length \(r\) tends to 0
  - If \(m \gg 2^r\), then prob. tends to 0
    - \((1 - 2^{-r})^m \approx e^{-m2^{-r}} = 0\) as \(m/2^r \to \infty\)
    - So, the probability of finding a tail of length \(r\) tends to 1

- Thus, \(2^R\) will almost always be around \(m!\)
E[2^R] is actually infinite
- Probability halves when \( R \rightarrow R+1 \), but value doubles

Workaround involves using many hash functions \( h_i \) and getting many samples of \( R_i \)

How are samples \( R_i \) combined?
- Average? What if one very large value \( 2^{R_i} \)?
- Median? All estimates are a power of 2

Solution:
- Partition your samples into small groups
- Take the median of groups
- Then take the average of the medians
(3) Computing 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 $k^{th}$ moment is

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

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

- **0th moment** = number of distinct elements
  - The problem just considered
- **1st moment** = count of the numbers of elements = length of the stream
  - Easy to compute
- **2nd moment** = surprise number $S$ = a measure of how uneven the distribution is
Example: Surprise Number

- **Stream of length 100**
- **11 distinct values**

- Item counts: \(10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9\)
  \(\text{Surprise } S = 910\)

- Item counts: \(90, 1, 1, 1, 1, 1, 1, 1, 1, 1\)
  \(\text{Surprise } S = 8,110\)
AMS Method

- AMS method works for all moments
- Gives an unbiased estimate
- We will just concentrate on the 2\textsuperscript{nd} moment $S$
- We pick and keep track of many variables $X$:
  - For each variable $X$ we store $X.el$ and $X.val$
    - $X.el$ corresponds to the item $i$
    - $X.val$ corresponds to the count of item $i$
  - Note this requires a count in main memory, so number of $X$s is limited
- Our goal is to compute $S = \sum_i m_i^2$
How to set $X.val$ and $X.el$?
- Assume stream has length $n$ (we relax this later)
- Pick some random time $t$ ($t < n$) to start, so that any time is equally likely
- Let at time $t$ the stream have item $i$. **We set $X.el = i$**
- Then we maintain count $c$ ($X.val = c$) of the number of $i$s in the stream starting from the chosen time $t$

Then the estimate of the 2nd moment ($\sum_i m_i^2$) is:

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

- Note, we will keep track of multiple $X$s, ($X_1, X_2, ... X_k$) and our final estimate will be $S = \frac{1}{k} \sum_j^k f(X_j)$
2nd moment is $S = \sum_i m_i^2$

$c_t$ ... number of times item at time $t$ appears from time $t$ onwards ($c_1=ma$, $c_2=ma-1$, $c_3=mb$)

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

$= \frac{1}{n} \sum_i n (1 + 3 + 5 + \cdots + 2m_i - 1)$

$m_i$ ... total count of item $i$ in the stream (we are assuming stream has length $n$)
**Expectation Analysis**

- **Count:** 1 2 3 4
  - Stream: a a b b a b
  - Then \( E[f(X)] = \frac{1}{n} \sum_i n \left( 1 + 3 + 5 + \cdots + 2m_i - 1 \right) \)
  - Little side calculation: \( (1 + 3 + 5 + \cdots + 2m_i - 1) = \sum_{i=1}^{m_i} (2i - 1) = 2 \frac{m_i(m_i+1)}{2} - m_i = (m_i)^2 \)

- Then \( E[f(X)] = \frac{1}{n} \sum_i n \left( (m_i)^2 \right) \)

- So, \( E[f(X)] = \sum_i (m_i)^2 = S \)
- We have the second moment (in expectation)!
For estimating $k^{th}$ moment we essentially use the same algorithm but change the estimate:

- For $k=2$ we used $n(2c - 1)$
- For $k=3$ we use: $n(3c^2 - 3c + 1)$ (where $c=X.val$)

Why?

- For $k=2$: Remember we had $(1 + 3 + 5 + \cdots + 2m_i - 1)$ and we showed terms $2c-1$ (for $c=1,\ldots,m$) sum to $m^2$
  - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c - 1)^2 = m^2$
  - So: $2c - 1 = c^2 - (c - 1)^2$
- For $k=3$: $c^3 - (c-1)^3 = 3c^2 - 3c + 1$

Generally: Estimate $= n\left(c^k - (c - 1)^k\right)$
In practice:
- Compute \( f(X) = n(2c - 1) \) for as many variables \( X \) as you 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
(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 $X$s 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

- $1 = \text{item present}; \ 0 = \text{not present}$
- Use **DGIM** to estimate counts of $1$s for all items
Extensions

- In principle, you 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

- Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)
  - 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 $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time $t$ to be:
    \[
    \sum_{i=1}^{t} a_i (1 - c)^{t-i}
    \]
    - $c$ is a constant, presumably tiny, like $10^{-6}$ or $10^{-9}$
  - When new $a_{t+1}$ arrives:
    Multiply current sum by $(1-c)$ and add $a_{t+1}$
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
  - That is: $\sum_{i=1}^{t} \delta_i \cdot (1 - c)^{t-i}$
    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 $x$
Important property: Sum over all weights 
\[ \sum_t (1 - c)^t \text{ is } 1/[1 - (1 - c)] = 1/c \]
What are “currently” most popular movies?

Suppose we want to find movies of weight > \( \frac{1}{2} \)

- **Important property**: Sum over all weights
  \[ \sum_t (1 - c)^t \] is \( \frac{1}{1 - (1 - c)} = \frac{1}{c} \)

- **Thus:**
  - There cannot be more than \( \frac{2}{c} \) movies with weight of \( \frac{1}{2} \) or more
  - So, \( \frac{2}{c} \) is a limit on the number of movies being counted at any time
Count (some) itemsets in an E.D.W.

What are currently “hot” itemsets?

Problem: 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 < \( \frac{1}{2} \)
- Initiate new counts (next slide)
Start a count for an itemset \( S \subseteq B \) if every proper subset of \( S \) had a count prior to arrival of basket \( B \)

- **Intuitively:** 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 \( \{i, j\}, \{i, k\}, \{j, k\} \) were all counted prior to seeing \( B \)
Counts for single items $< (2/c) \cdot (\text{avg. number of items in a basket})$

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 $1\text{M}$ counts