Map Reduce and Streaming Calculations

Giri Iyengar

Cornell University

gi43@cornell.edu

April 18, 2018

Agenda for the week

- Map-Reduce
- Poisson resampling
- Streaming Calculations
  1. Reservoir Sampling
  2. Storing Items in Sets
  3. Counting in single pass
  4. Frequent Items in a stream
  5. Estimating CDF/PDF in streaming mode
- Background Reading

Overview

Why Streaming Calculations

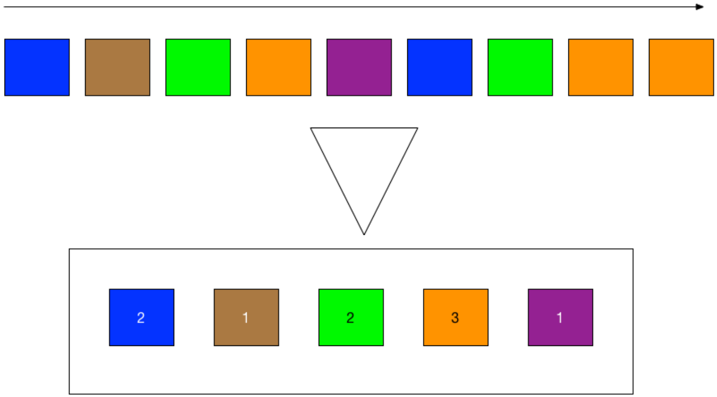
Streaming Calculations

In computer science, streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). These algorithms have limited memory available to them (much less than the input size) and also limited processing time per item. These constraints may mean that an algorithm produces an approximate answer based on a summary or "sketch" of the data stream in memory.

Growing Data sizes relative to processing capacity

Our ability and desire to collect and process data is growing faster than our ability to store, process, analyze, compute, and reason with data. At the same time, evidence-based reasoning is very critical for business success, more so now than ever before!

Streaming Calculations



Big Data makes even simple things painful

Even simple things like histograms, unique counts become hard when there is too much data.

Reservoir Sampling

Sample properly from a stream

Potentially infinite stream of data - how do we extract a representative sample from it?

Reservoir Sampling

- Fast algorithms for selecting n items without replacement from a pool of N records where N is not known in advance
- Sampling in one pass using **constant space** in optimum time

Reservoir Sampling

Reservoir Algorithms

First step in all such algorithms is to maintain a reservoir of size n where the first n records are placed. After that, all subsequent records are either stored in the reservoir by kicking some sample out or skipped. At any point in time, the state of the reservoir represents a random sample of the original streaming dataset.

Reservoir Sampling

- Fill the reservoir with first n samples
- For sample $t+1, (t \geq n)$, accept with probability $\frac{n}{t+1}$, reject otherwise
- If accepted, randomly evict one of the existing n samples and replace with the new sample

Variations of Reservoir Sampling

- Do we examine every subsequent sample?
- Instead, can we skip a few records?
- If we skip, how do we perform the skip?

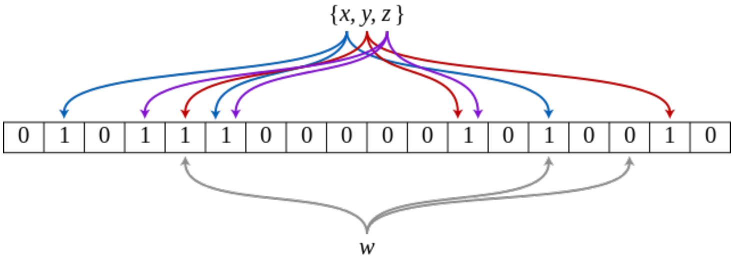
Storing Items in Sets

Bloom Filter

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, thus a Bloom filter has a 100% recall rate. In other words, a query returns either *"possibly in set"* or *"definitely not in set"*.

- Elements can be added to the set
- Once added, Elements cannot be removed
- More Elements in the set, greater the false-positive rate

Bloom Filters



m - bit vector size. k - number of independent hash functions.

- Take each element. Hash it k times
- Each hash will set one of the bits in the m vector to 1. Each element will set k bits to 1
- During query time, use the same k hash functions. If all hashes match, the item maybe in the set. If not, definitely not there

Bloom Filter Details

- For m bit vector, probability of a bit not set to 1 by a single hash function is $1 - \frac{1}{m}$
- With k hash functions, probability of a bit not set to 1 is $(1 - \frac{1}{m})^k$
- With n items in Filter, probability that a bit is 0 is $(1 - \frac{1}{m})^{kn}$
- Probability of a bit being 1 is $1 - (1 - \frac{1}{m})^{kn}$
- Probability of k bits being 1 is $(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k$

Filter Design

Given n and a false-positive rate you can tolerate, you can compute optimal values of k and m.

Extensions to Bloom Filters

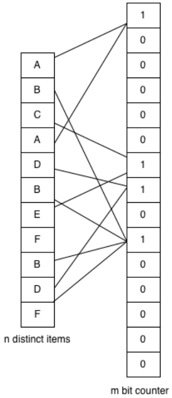
- Counting Bloom Filters – allow delete operations
- Bloomier Filters – associative arrays that return values of keys
- Scalable Bloom Filters – keeps false positive rate under check as n grows
- Stable Bloom Filters – adaptation to streaming data by evicting items
- Layered Bloom Filters – can remember the number of times an item was seen

Counting Number of Unique items in a Single Pass

Problem Statement

In a potentially infinite stream, how many unique items have we seen so far? Bloom Filters can tell you whether an item was seen on not. We also want to know how many types of items we saw so far.

Linear Counting



Example

- Estimated Linear Count is $\hat{n} = -m \ln V_n, V_n = \frac{U_n}{m}$
- U_n is the number of empty bins
- For $m = 15, n = 6$, let's say we got this bit vector after hashing. We estimate the count to be 4.65 items.

Linear Counting

Sketch of Linear Count estimate

- Let A_j be the event that bin j is empty
- Since all assignments are independent, $P(A_j) = (1 - \frac{1}{m})^n$
- $E(U_n) = \sum_{j=1}^{m} P(A_j) = m(1 - \frac{1}{m})^n \approx me^{-\frac{n}{m}}$
- $E(V_n) = E(\frac{U_n}{m}) \approx e^{-\frac{n}{m}}$
- $\therefore \hat{n} = -m \ln V_n$

FM Sketch algorithm or LogLog Counting

LogLog Counting

- Any number can be represented as $y = \sum_{k \geq 0} bit(y,k)2^k$
- Let $\rho(y) = \min_{k \geq 0} bit(y,k) \neq 0$
- Given a set of M items with n distinct, hash one by one and compute $\rho(hash(x)), \forall x \in M$
- In a bit vector, set the corresponding bit to 1
- Find the most significant bit that is 1. Let it be R. We have seen approximately $\frac{2^R}{\phi}$ items, where ϕ is an empirically determined correction factor

Intuition

If we store n items, the least significant bit is accessed $n/2$ times, the next bit is accessed $n/4$ times, and so forth

HyperLogLog

Problem with LogLog

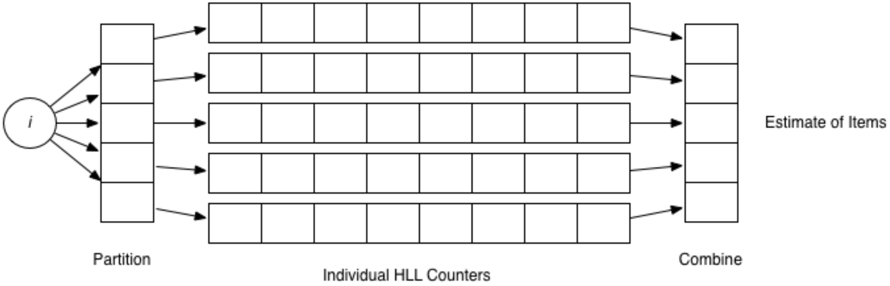
Not that stable. Can be improved by doing several hashes and taking mean/median as the estimated count

HyperLogLog Intuition

The number of distinct items can be estimated by observing the number of leading zeros in the binary representation of a number

- Simple estimate of cardinality with HLL intuition above leads to large variance
- Minimize variance by splitting into subsets, and come up with independent estimates for each subset
- Final cardinality estimate is harmonic mean of each subset estimate

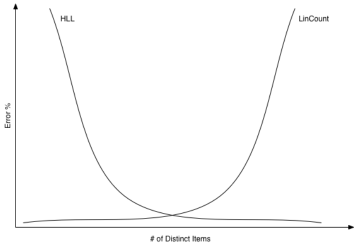
HyperLogLog



Putting it all together: HLL++

Linear Counting vs HLL

- LinCount Is fairly accurate for low cardinalities
- HLL is not that accurate for low cardinalities
- HLL++: Start with LinCount and then switch to HLL



Counting Frequent items in a Stream

- Frequent(k)
- LossyCounting(k)
- SpaceSaving(k)

Frequent(k)

Sketch of the algorithm

1. Initialize storage T of size k
2. For all i
   - If $i \in T$, $c_i = c_i + 1$
   - Else if $|T| < k - 1$, $T = T \cup \{i\}, c_i = 1$
   - Else $\forall j \in T$, $c_j = c_j - 1$
   - If $c_j == 0$, evict j from T

Any item that occurs more than n/k times can be shown to appear in this register. However, the frequency counts of the items are not accurate

LossyCounting(k)

Algorithm Sketch

1. Initialize storage T of size k
2. For all i
    - $\Delta(i) = \lfloor \frac{i}{k} \rfloor$.
    - If $i \in T$, $c_i = c_i + 1$
    - Else if $|T| < k$, $T = T \cup \{i\}, c_i = 1$
    - Eliminate entry j if $c_j + \Delta(j) < \Delta(i)$

In this counting approach, Δ gives you the lower bound of count of an item. So, the actual count for an item is $c_i + \Delta$. For a given error ϵ, if you choose $k = \frac{1}{\epsilon}$, you are guaranteed that all counts are off by at-most ϵn.

SpaceSaving(k)

Sketch of the algorithm

1. Initialize storage T of size k
2. For all i
    - If $i \in T$, $c_i = c_i + 1$
    - Else if $|T| < k - 1$, $T = T \cup \{i\}, c_i = 1$
    - Else remove item $\arg\min_j c_j$
    - Replace item with least count with (i, c_{min})

Both LossyCounting and SpaceSaving have highly accurate counts for items stored early and not removed. Items stored later in the stream have increasing error.

Overview

Reading

- HLL++
- topK
- Hyper Parameter Optimization
- Calibration

thank you
danke
謝謝
ngiyabonga
teşekkür ederim
gracias
спасибо
Баярлалаа
рахмат
bedankt
hvala
mauruuru
dziękuję
obrigado
sukriya
kop khun krap
go raibh maith agat
mochchakkeram
tapadh leat
dank je
merci
grazie
arigatō
dakujem
мерси
terima kasih
감사합니다
ありがとう
ευχαριστώ
takk
taiku
sagolun
faafetai lava
kiitos
dankie
dhanyavad
nanni
nandri
bayarlalaa
gracie
köszönöm
enkosi
sobodi
děkuji
mesi
didi madloba
kam sah hamnida
najis tuke
rahmat
chnorakaloutioun
gratias ago
gracies
sulpáy
tanemirt
rahmet
diolch
dhanyavadagalu
shukriya
merce
trugarez
mamnun
дякую
djiere dieuf
tau
chokrane
murakoze
tenki
obrigada
asante
manana
хвала
mahalo
paldies
grazzi
matondo
misaotra
blagodaram
спасибі
vinaka
mersi
kia ora
barka
welalin
tack
spas
dankon
aciū
akun
xièxie