More Stream Mining

Bloom Filters
Sampling Streams
Counting Distinct Items
Computing Moments

Cloud and Big Data Summer School, Stockholm, Aug., 2015
Jeffrey D. Ullman

Filtering Stream Content

- To motivate the Bloom-filter idea, consider a web crawler.
- It keeps, centrally, a list of all the URL’s it has found so far.
- It assigns these URL’s to any of a number of parallel tasks; these tasks stream back the URL’s they find in the links they discover on a page.
- It needs to filter out those URL’s it has seen before.

Role of the Bloom Filter

- A Bloom filter placed on the stream of URL's will declare that certain URL's have been seen before.
- Others will be declared new, and will be added to the list of URL's that need to be crawled.
- Unfortunately, the Bloom filter can have false positives.
  - It can declare a URL has been seen before when it hasn't.
  - But if it says "never seen," then it is truly new.

How a Bloom Filter Works

- A *Bloom filter* is an array of bits, together with a number of hash functions.
- The argument of each hash function is a stream element, and it returns a position in the array.
- Initially, all bits are 0.
- When input x arrives, we set to 1 the bits h(x), for each hash function h.

Example: Bloom Filter

- Use N = 11 bits for our filter.
- Stream elements = integers.
- Use two hash functions:
  - $h_1(x)$ =
    - Take odd-numbered bits from the right in the binary representation of x.
    - Treat it as an integer i.
    - Result is i modulo 11.
  - $h_2(x)$ = same, but take even-numbered bits.

Example – Continued

Stream element	h_1	h_2	Filter contents
			00000000000
25 = 11001	5	2	00100100000
159 = 10011111	7	0	10100101000
585 = 1001001001	9	7	10100101010

Bloom Filter Lookup

- Suppose element y appears in the stream, and we want to know if we have seen y before.
- Compute h(y) for each hash function y.
- If all the resulting bit positions are 1, say we have seen y before.
- If at least one of these positions is 0, say we have not seen y before.

Example: Lookup

- Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010.
- Lookup element y = 118 = 1110110 (binary).
- $h_1(y)$ = 14 modulo 11 = 3.
- $h_2(y)$ = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are sure y is not in the set.

Performance of Bloom Filters

- Probability of a false positive depends on the density of 1's in the array and the number of hash functions.
  - = (fraction of 1's)$^{\text{\# of hash functions}}$.
- The number of 1's is approximately the number of elements inserted times the number of hash functions.
  - But collisions lower that number slightly.

Throwing Darts

- Turning random bits from 0 to 1 is like throwing *d* darts at *t* targets, at random.
- How many targets are hit by at least one dart?
- Probability a given target is hit by a given dart = 1/t.
- Probability none of d darts hit a given target is $(1-1/t)^d$.
- Rewrite as $(1-1/t)^{t(d/t)}$ ~= $e^{-d/t}$.

Example: Throwing Darts

- Suppose we use an array of 1 billion bits, 5 hash functions, and we insert 100 million elements.
- That is, $t = 10^9$, and $d = 5*10^8$.
- The fraction of 0's that remain will be $e^{-1/2} = 0.607$.
- Density of 1's = 0.393.
- Probability of a false positive = $(0.393)^5 = 0.00937$.

Sampling a Stream

What Doesn't Work
Sampling Based on Hash Values

When Sampling Doesn't Work

- Suppose Google would like to examine its stream of search queries for the past month to find out what fraction of them were unique – asked only once.
- But to save time, we are only going to sample 1/10th of the stream.
- The fraction of unique queries in the sample != the fraction for the stream as a whole.
  - In fact, we can't even adjust the sample's fraction to give the correct answer.

Example: Unique Search Queries

- The length of the sample is 10% of the length of the whole stream.
- Suppose a query is unique.
  - It has a 10% chance of being in the sample.
- Suppose a query occurs exactly twice in the stream.
  - It has an 18% chance of appearing exactly once in the sample.
- And so on … The fraction of unique queries in the stream is unpredictably large.

Sampling by Value

- Our mistake: we sampled based on the position in the stream, rather than the value of the stream element.
- The right way: hash search queries to 10 buckets 0, 1,…, 9.
- Sample = all search queries that hash to bucket 0.
  - All or none of the instances of a query are selected.
  - Therefore the fraction of unique queries in the sample is the same as for the stream as a whole.

Controlling the Sample Size

- Problem: What if the total sample size is limited?
- Solution: Hash to a large number of buckets.
- Adjust the set of buckets accepted for the sample, so your sample size stays within bounds.

Example: Fixed Sample Size

- Suppose we start our search-query sample at 10%, but we want to limit the size.
- Hash to, say, 100 buckets, 0, 1,…, 99.
  - Take for the sample those elements hashing to buckets 0 through 9.
- If the sample gets too big, get rid of bucket 9.
- Still too big, get rid of 8, and so on.

Sampling Key-Value Pairs

- This technique generalizes to any form of data that we can see as tuples (K, V), where K is the "key" and V is a "value."
- Distinction: We want our sample to be based on picking some set of keys only, not pairs.
  - In the search-query example, the data was "all key."
- Hash keys to some number of buckets.
- Sample consists of all key-value pairs with a key that goes into one of the selected buckets.

Example: Salary Ranges

- Data = tuples of the form (EmpID, Dept, Salary).
- Query: What is the average range of salaries within a department?
- Key = Dept.
- Value = (EmpID, Salary).
- Sample picks some departments, has salaries for all employees of that department, including its min and max salaries.

Counting Distinct Elements

Applications
Flajolet-Martin Approximation Technique
Generalization to Moments

Counting Distinct Elements

- Problem: a data stream consists 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.

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 unique users visited Facebook this month?
- How many different pages link to each of the pages we have crawled.
  - Useful for estimating the PageRank of these pages.

Estimating Counts

- Real Problem: what if we do not have space to store the complete set?
- Estimate the count in an unbiased way.
- Accept that the count may be in 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 0's in $h(a)$.
- Record R = the maximum $r(a)$ seen.
- Estimate = 2^R.

Why It Works

- The probability that a given $h(a)$ ends in at least i 0's is 2^{-i}.
- If there are m different elements, the probability that $R \geq i$ is $1 - (1 - 2^{-i})^m$.

Prob. all h(a)'s end in fewer than i 0's.

Prob. a given h(a) ends in fewer than i 0's.

Why It Works – (2)

- Since 2^{-i} is small, $1 - (1-2^{-i})^m \approx 1 - e^{-m2^{-i}}$.
- If $2^i >> m$, $1 - e^{-m2^{-i}} \approx 1 - (1 - m2^{-i}) \approx m/2^i \approx 0$.
- If $2^i << m$, $1 - e^{-m2^{-i}} \approx 1$.
- Thus, 2^R will almost always be around m.

First 2 terms of the
Taylor expansion of e^x

Why It Doesn't Work

- $E(2^R)$ is, in principle, infinite.
  - Probability halves when R -> R+1, but value doubles.
- Workaround involves using many hash functions and getting many samples.
- How are samples combined?
  - Average? What if one very large value?
  - Median? All values are a power of 2.

Solution

- Partition your samples into small groups.
  - O(log n), where n = size of universal set, suffices.
- Take the average within each group.
- Then take the median of the averages.

Generalization: Moments

- Suppose a stream has elements chosen from a set of n values.
- Let m_i be the number of times value i occurs.
- The k^{th} *moment* is the sum of $(m_i)^k$ over all i.

Special Cases

- 0^{th} moment = number of different elements in the stream.
  - The problem just considered.
- 1^{st} moment = count of the numbers of elements = length of the stream.
  - Easy to compute.
- 2^{nd} moment = *surprise number* = a measure of how uneven the distribution is.

Example: Surprise Number

- Stream of length 100; 11 values appear.
- Unsurprising: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9. Surprise # = 910.
- Surprising: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1. Surprise # = 8,110.

AMS Method

- Works for all moments; gives an unbiased estimate.
- We'll just concentrate on 2^{nd} moment.
- Based on calculation of many random variables *X*.
  - Each requires a count in main memory, so number is limited.

One Random Variable

- Assume stream has length *n*.
- Pick a random time to start, so that any time is equally likely.
- Let the chosen time have element *a* in the stream.
- *X* = *n* * ((twice the number of *a*'s in the stream starting at the chosen time) – 1).
  - Note: store *n* once, count of *a*'s for each *X*.

Expected Value of X

- 2^{nd} moment is $\Sigma_a (m_a)^2$.
- $E(X) = (1/n)\left(\Sigma_{\text{all times } t}\ n * \text{(twice the number of times the stream element at time } t \text{ appears from that time on)} - 1\right)$.
- $= \Sigma_a (1/n)(n)(1+3+5+\ldots+2m_a-1)$.
- $= \Sigma_a (m_a)^2$.

Group times by the value seen

Time when the last a is seen

Time when penultimate a is seen

Time when the first a is seen

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* changes; it is the number of inputs seen so far.

Fixups

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 cannot have one random variable X for each start-time, and must throw out some start-times as we read the stream.
   - Objective: each starting time *t* is selected with probability *k*/*n*.

Solution to (2)

- 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 out, with equal probability.
- Probability of each of the first n-1 positions being chosen:

$$(n-k)/n * k/(n-1) + k/n * k/(n-1) * (k-1)/k = k/n$$

$(n-k)/n$	$k/(n-1)$	k/n	$k/(n-1)$	$(k-1)/k$
n-th position not chosen	Previously chosen	n-th position chosen	Previously chosen	Survives