MPEI 2018-2019

#22 – Bloom Filters (continuation)

Analysis

Basic idea: Throwing Darts

- 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

Probabilities for one bit

- Initially all bits are set to zero.

- What is the probability of bit $b_i = 1$ after using the first hash function when inserting an element ?

- Assuming hash function selects each array position with equal probability, is $1/n$
- So, the probability of $b_i = 0$ is $1 - \frac{1}{n}$

- To insert the element this process is repeated k times

- Resulting in probability of having $b_i = 0$:

$$\left(1 - \frac{1}{n}\right)^k$$

Probabilities for one bit (cont.)

- Since there are *k* hash functions and *m* elements to insert in the filter, the probability of having $b_i = 0$ after handling all *m* elements (assuming independence) is:
- $\left(1 - \frac{1}{n}\right)^{k\,m}$
- $= \left[\left(1 - \frac{1}{n}\right)^{m}\right]^{k} = \ a^{k}$
- The probability of $b_i = 1$ is $\ 1 - a^{k}$
- Related to the **probability that a target gets at least one dart**

False positives and false negatives

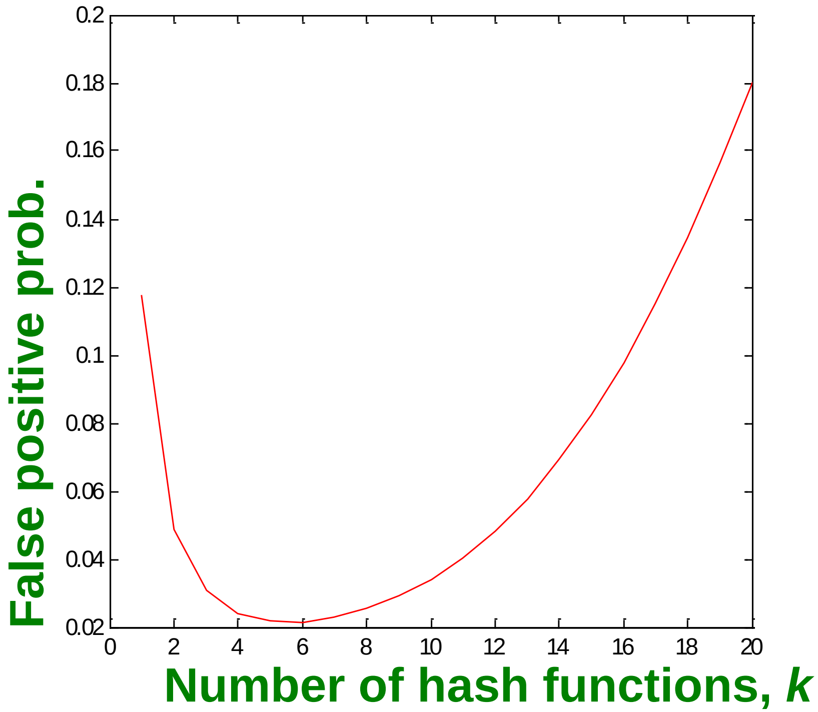
Element in set ?	Result of membership test	Correct ?	Type of error
Yes	Yes (in set)	Yes	
	No	Error	False negative
No	Yes	Error	False positive
	No	Yes	

Probability of a false positive

- There is a false positive error when we look up a bit string that is not in *S*, but find the corresponding *k* bits set to 1
- The probability of such event, after inserting *m* elements, is:
- $p = \left[1 - \left(1 - \frac{1}{n}\right)^{k\,m}\right]^k = \left(1 - a^k\right)^k$
- $p \approx \left(1 - e^{-km/n}\right)^k$
  - Using the approximation $(1-\epsilon)^{1/\epsilon} = 1/e$ (small ϵ)

Effect of 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**



- **Adding a second hash function reduced false positives**

- **What happens as we keep increasing *k*?**

"Optimal" value of k

- To determine the number of hash functions (*k*) that minimizes the probability of error (*p*) we minimize $\log\ p$, more tractable $[\log = \ln]$

- $\log\ p = k\ \ \log\ (1 - a^k)$
- Finding derivative and setting to zero:
- $\left(1 - a^k\right) \log\ \left(1 - a^k\right) - a^k \log a^k = 0$
- With solution $a^k = \frac{1}{2}$

- The best value of *k* is therefore
- $k_{opt} = \frac{\log 1/2}{\log a} = \frac{\log 1/2}{m \log(1 - \frac{1}{n})} \approx \frac{n \log 2}{m} \approx \frac{0.693\ n}{m}$

"Optimal" value of k

- As k_{opt} is not an integer, in practice we use the closest integer

- "Optimal" value of ***k***: ***n/m*** **ln(2)**

- **In our example (*m* = 1 billion, *n* = 8 billion)**
  - Optimal **k = 8 ln(2) = 5.54 ≈ 6**

Lower bound for the error probability

- If we could have $a^k = 1/2$ for an integer k_{opt}, the equation $\left(1 - a^k\right)^k$ would lead to:

- $p_{opt} = \left(1 - \frac{1}{2}\right)^{k_{opt}}$
- $= \left(\frac{1}{2}\right)^{k_{opt}}$
- $= 2^{-k_{opt}}$

- That can be taken as the lower bound for the error probability (false positives)

Value for n ?

- The required *n*, given *m* and a desired false positive probability *p* - and assuming the optimal value of *k* is used - can be computed by substituting the optimal value of *k* [=***n/m*** **ln(2)**] in the probability expression

  $$p \approx \left(1 - e^{-km/n}\right)^k$$

- n= ?
  - Homework

One or more arrays ?

- Is it better to have **1** big **B** or ***k*** small **B**s?
- $(1 - e^{-km/n})^k$ vs $(1 - e^{-m/(n/k)})^k$
- **It is the same**
- But keeping **1 big B** is simpler

Perfect Hashing

- If the membership set is known in advance then better performance than with a standard Bloom Filter can be achieved using related techniques.

- For instance we can establish an arbitrarily low false positive rate by using perfect hash functions and limiting the size of the shared hash table.

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

Trade-offs

- The error rate can be decreased
  - by increasing the number of hash functions
  - and the space allocated to store the table

Final remarks

- Bloom Filters should be considered for programs where an imperfect set membership test could be helpfully applied to a large data set of unknown composition

- The great advantage of Bloom Filters over the use of single hash transforms is their speed and set error rate.

- Although the method can be applied to sets of any size, small sets are better dealt with by trees and heaps

Counting Filters

The origin

- A basic Bloom filter can only represent a set, but neither allows for querying the multiplicities of an item, nor does it support deleting entries
- The term *counting Bloom filter* is used to refer to variants of Bloom filters that represent multisets rather than sets
- Technically, a counting Bloom filter extends a basic Bloom filter with width parameter w (bits).

The basic idea

- To create a Counting filter the array positions are extended from a single bit to being an w bit counter.

- Counting filters provide a way to implement a *delete* operation on a Bloom filter without recreating the filter afresh
  - The original counting Bloom filter used cells with w=4 only to support deletion, not to count elements
    - Making counting Bloom filters to use 3 to 4 times more space than basic Bloom filters.

Changes to insert() and isMember()

- The insert operation is extended to *increment* the value of the buckets
- The lookup operation checks that each of the required buckets is non-zero.

delete()

- New operation
- The delete operation consists of decrementing the value of each of the respective buckets
- **Deletions necessarily introduce false negative (FN) errors**
  - when you flip a set bit back to 0 that was part of a k bits from another item, the Bloom filter will no longer report that item as belonging to the set

count()

- New operation

- Retrieving the count of an item of the set involves computing its set of counters and returning the minimum value as frequency estimate

  - This query algorithm is also known as *minimum selection* (MS).

Issues

- There exist two main issues with counting Bloom filters:
- Counter overflows
  - exists when the counter value reaches $2w-1$ and cannot be incremented anymore.
  - One typically stops counting as opposed to overflowing and restarting at 0.
  - However, this strategy introduces ***undercounts***, which we also refer to as FNs.
- The choice of w
  - A large w quickly diminishes the space savings from using of a Bloom filter
    - There will also be a lot of unused space manifesting as unused zeros.
  - A small w may quickly lead to maximum counter values
  - Choosing the right value is a difficult trade-off that depends on the distribution of the data

Limitations

- Counting filters have limited scalability
- Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance.

- Once the designed capacity of the table is exceeded, the false positive rate will grow rapidly as more keys are inserted

Matlab

- Adapt insert and isMember
- Create the new functions:
  - delete()
  - count()
- Create a simple test

Links used

- http://matthias.vallentin.net/blog/2011/06/a-garden-variety-of-bloom-filters/
- https://en.wikipedia.org/wiki/Bloom_filter#Counting_filters

Other techniques

- Variants of Bloom
  - See, for example:
  - http://matthias.vallentin.net/course-work/cs270-s11.pdf

- Cuckoo Hashing
  - See:
  - http://www.lkozma.net/cuckoo_hashing_visualization/

Part of the slides adapted from: Mining Data Streams

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
http://www.mmds.org