

Bloom filters and Min Hash for Information Retrieval

EECS 395/495 Fall 2012

Doug Downey

Motivation

- Duplicate pages (also: urls for url frontier)
 - Save and compare hashes
- *Near*-duplicate pages
 - Tougher

Motivation

- Duplicate pages (also: urls for url frontier)
 - Save and compare hashes
(still large – bloom filters)
- *Near*-duplicate pages
 - Tougher

Bloom Filters

- Whenever you have a set or list, and space is an issue, a Bloom filter may be a useful alternative.

The main point

- Whenever you have a set or list, and space is an issue, a Bloom filter may be a useful alternative.

The Problem Solved by BF: Approximate Set Membership

- Given a set $S = \{x_1, x_2, \dots, x_n\}$, construct data structure to answer queries of the form “Is y in S ?”
- Data structure should be:
 - **Fast** (Faster than searching through S).
 - **Small** (Smaller than explicit representation, simple hash).
- To obtain speed and size improvements, allow some probability of error.
 - **False positives**: $y \notin S$ but we report $y \in S$
 - **False negatives**: $y \in S$ but we report $y \notin S$

Bloom Filters

Start with an m bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Hash each item x_j in S k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

To check if y is in S , check B at $H_i(y)$. All k values must be 1.

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Possible to have a false positive; all k values are 1, but y is not in S .

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

n items

$m = cn$ bits

k hash functions

False Positive Probability

- Pr(specific bit of filter is 0) is

$$p' \equiv (1 - 1/m)^{kn} \approx e^{-kn/m} \equiv p$$

- If ρ is fraction of 0 bits in the filter then false positive probability is

$$(1 - \rho)^k \approx (1 - p')^k \approx (1 - p)^k = (1 - e^{-k/c})^k$$

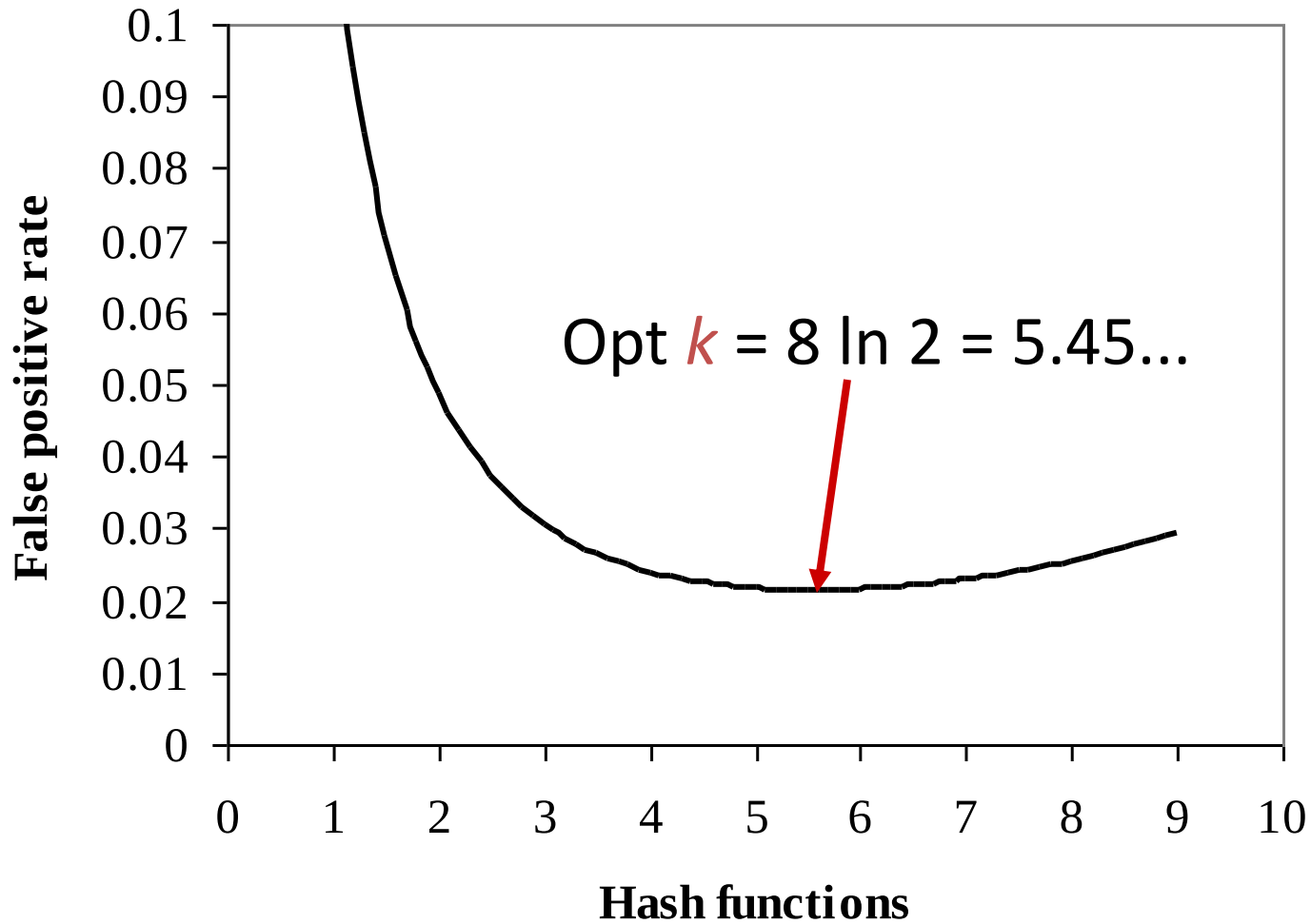
- Find optimal at $k = (\ln 2)m/n$ by calculus.
 - So optimal fpp is about $(0.6185)^{m/n}$

n items

$m = cn$ bits

k hash functions

Example



$$m/n = 8$$

n items

$m = cn$ bits

k hash functions

Classic Uses of BF: Spell-Checking

- Once upon a time, memory was scarce...
- **/usr/dict/words** -- about 210KB, 25K words
- Use 25 KB Bloom filter
 - 8 bits per word.
 - Optimal 5 hash functions.
- Probability of false positive about 2%
- False positive = accept a misspelled word
- BFs still used to deal with list of words
 - Password security [Spafford 1992], [Manber & Wu, 94]
 - Keyword driven ads in web search engines, etc

Classic Uses of BF: Data Bases

- **Join:** Combine two tables with a common domain into a single table
- **Semi-join:** A join in distributed DBs in which only the joining attribute from one site is transmitted to the other site and used for selection. The selected records are sent back.
- **Bloom-join:** A semi-join where we send only a BF of the joining attribute.

Example

Empl	Salary	Addr	City
John	60K	...	New York
George	30K	...	New York
Moe	25K	...	Topeka
Alice	70K	...	Chicago
Raul	30K		Chicago

City	Cost of living
New York	60K
Chicago	55K
Topeka	30K

- Create a table of all employees that make $< 40K$ and live in city where COL $> 50K$.

Empl	Salary	Addr	City	COL
------	--------	------	------	-----

- **Join:** send (City, COL) for COL > 50 . **Semi-join:** send just (City).
- **Bloom-join:** send a Bloom filter for all cities with COL > 50

Motivation

- Duplicate pages (also: urls for url frontier)
 - Save and compare hashes
(still large – bloom filters)
- *Near*-duplicate pages
 - Tougher
 - MinHash

What about near-duplicates?

- Shingle – contiguous subsequence of w tokens
 - a.k.a. w -gram
- Idea: docs are near duplications iff they share a high proportion of shingles
- Easy to compute proportion of shingles in common for two docs
 - But, we don't want to *store* all the docs, or do all comparisons...

Idea: MinHash

- Associate each shingle in a doc with an ID in $\{0, \dots, L\}$
- Generate random permutations of the singles
e.g.,
 - 0 -> 423
 - 1 -> 2,102,302
 - 2 -> 403,230
 - ...

Key property

- Probability two random permutations match in their minimum (lowest id) element is...

$$\frac{|A \cap B|}{|A \cup B|}$$