Computing scores in a complete search system

Chapter 7 - IIR

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri

Quick Recap

Term Frequency Weight

- The log frequency weight of term *t* in a document *d* is defined as follows

$$w_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Inverse Document Frequency - IDF

- The document frequency df_t is defined as the number of documents that t occurs in.
- We define the idf weight of term t as follows:

$$\mathrm{idf}_t = \log_{10} \frac{N}{\mathrm{df}_t}$$

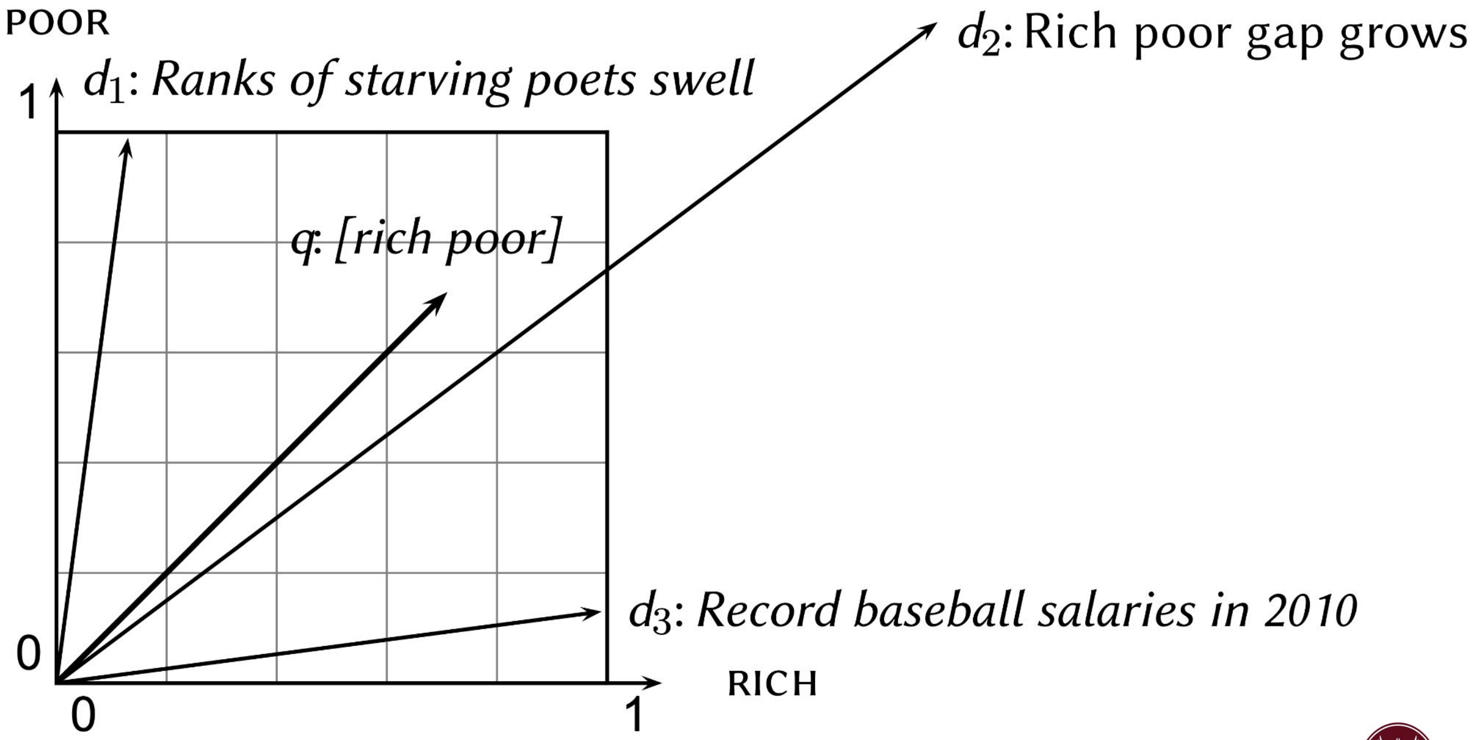
- idf is a measure of the informativeness of the term.

TF-IDF Weight

- The tf-idf weight of a term is the product of its tf weight and its idf weight:

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

Issue with Euclidean Distance



Cosine Similarity between Query and Document

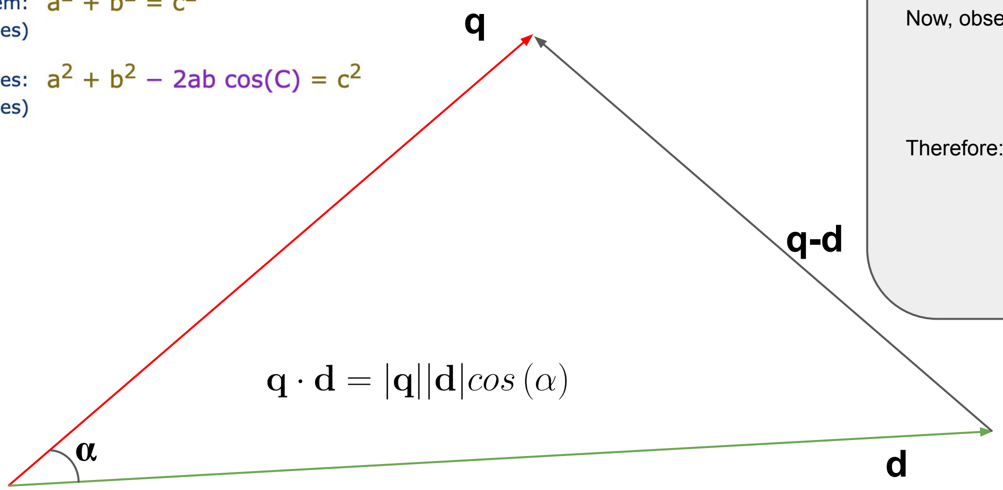
$$\cos(\vec{q}, \vec{d}) = \textsc{sim}(\vec{q}, \vec{d}) = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} = \sum_{i=1}^{|V|} \frac{q_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}} \cdot \frac{d_i}{\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- q_i is the tf-idf weight of term *i* in the query.
- d_i is the tf-idf weight of term *i* in the document.
- |***q***| and |***d***| are the lengths of vectors ***q*** and ***d****, respectively*.
- ***q***/|***q***| and ***d***/|***d***| are length-1 vectors (= normalized)

Cosine Similarity between Query and Document

Pythagoras Theorem: $a^2 + b^2 = c^2$
(only for Right-Angled Triangles)

Law of Cosines: $a^2 + b^2 - 2ab\cos(C) = c^2$
(for all triangles)



$$\mathbf{q} \cdot \mathbf{d} = |\mathbf{q}||\mathbf{d}|cos\,(\alpha)$$

By the Law of Cosines:

$$|\mathbf{q} - \mathbf{d}|^2 = |\mathbf{d}|^2 + |\mathbf{q}|^2 - 2|\mathbf{d}||\mathbf{q}|\cos(\alpha)$$

Now, observe that:

$$\begin{aligned} |\mathbf{q} - \mathbf{d}|^2 &= (\mathbf{q} - \mathbf{d}) \cdot (\mathbf{q} - \mathbf{d}) = \\ &= \mathbf{q} \cdot \mathbf{q} - 2(\mathbf{q} \cdot \mathbf{d}) + \mathbf{d} \cdot \mathbf{d} = \\ &= |\mathbf{q}|^2 + |\mathbf{d}|^2 - 2(\mathbf{q} \cdot \mathbf{d}) \end{aligned}$$

Therefore:

$$|\mathbf{d}|^2 + |\mathbf{q}|^2 - 2|\mathbf{d}||\mathbf{q}|\cos(\alpha) = |\mathbf{q}|^2 + |\mathbf{d}|^2 - 2(\mathbf{q} \cdot \mathbf{d})$$

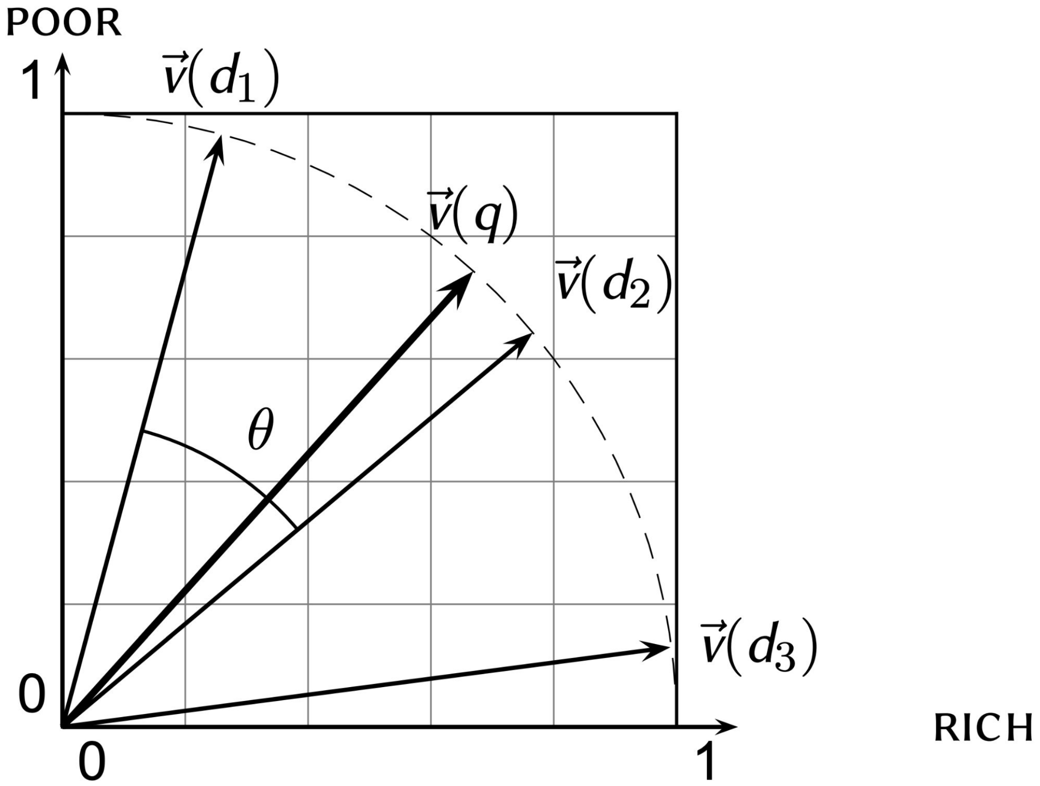
$$\Rightarrow$$

$$2|\mathbf{d}||\mathbf{q}|\cos(\alpha) = 2(\mathbf{q} \cdot \mathbf{d})$$

$$\Rightarrow$$

$$\mathbf{q} \cdot \mathbf{d} = |\mathbf{d}||\mathbf{q}|\cos(\alpha)$$

Length Normalization



Cosine Similarity Illustrated

- Query: “best car insurance”. Document: “car insurance auto insurance”

word	query					document			product
	tf-raw	tf-wght	df	idf	tf-idf weight	tf-raw	tf-wght	n’lized	
auto	0	0	5000	2.3	0	1	1	0.52	0
best	1	1	50000	1.3	1.3	0	0	0	0
car	1	1	10000	2.0	2.0	1	1	0.52	1.04
insurance	1	1	1000	3.0	3.0	2	1.3	0.68	2.04

$$\sqrt{1^2 + 0^2 + 1^2 + 1.3^2} \approx 1.92$$
$$1/1.92 \approx 0.52$$
$$1.3/1.92 \approx 0.68$$

- Final similarity score between query and document:

$$\sum_i w_{qi} \cdot w_{di} = 0 + 0 + 1.04 + 2.04 = 3.08$$

Is Cosine Enough?

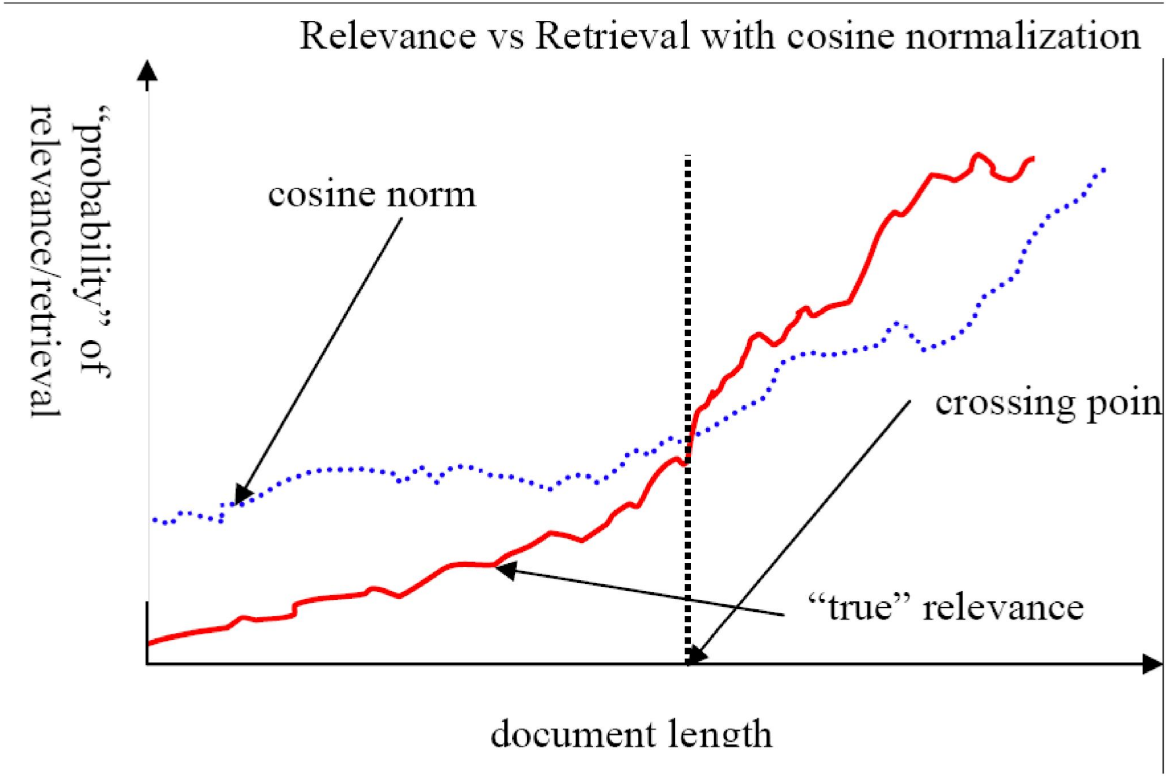
- Query q: “anti-doping rules Beijing 2008 olympics”
- Compare three documents
  - d1 → a short document on anti-doping rules at 2008 Olympics
  - d2 → a long document that consists of a copy of d1 and 5 other news stories, all on topics different from Olympics/anti-doping
  - d3 → a short document on anti-doping rules at the 2004 Athens Olympics
- What ranking do we expect in the vector space model?
  - d2 is likely to be ranked below d3 …
  - …but d2 is more relevant than d3.

Can we do something about this?

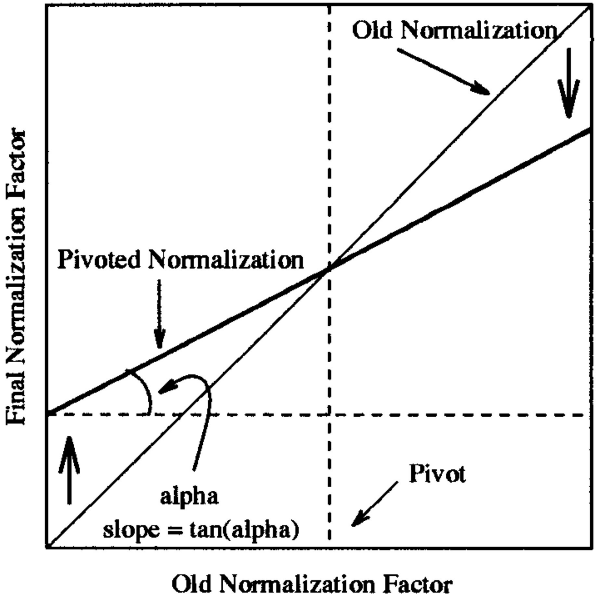
Pivot Normalization

- Cosine normalization produces weights that are too large for short documents and too small for long documents (on average).
- Adjust cosine normalization by linear adjustment: “turning” the average normalization on the pivot
- Effect: Similarities of short documents with query decrease; similarities of long documents with query increase.
- This removes the unfair advantage that short documents have.
- Note that “pivoted” scores are no longer bounded by 1.

Predicted and true probability of relevance



Pivot normalization



- pivoted normalization = (1.0 – slope) x pivot + slope x old normalization

Effect on Effectiveness: Amit Singhal's experiments

Cosine	Pivoted Cosine Normalization				
	Slope				
	0.60	0.65	0.70	**0.75**	0.80
6,526 0.2840	6,342 0.3024	6,458 0.3097	6,574 0.3144	**6,629** **0.3171**	6,671 0.3162
Improvement	+ 6.5%	+ 9.0%	+10.7%	**+11.7%**	+11.3%

- (relevant documents retrieved and (change in) average precision)

SAPIENZA
UNIVERSITÀ DI ROMA

Ranking

The Importance of Ranking

- Users want to look at a few results – not thousands.
- It’s very hard to write queries that produce a few results.
- Even for expert searchers
- → Ranking is important because it effectively reduces a large set of results to a very small one.

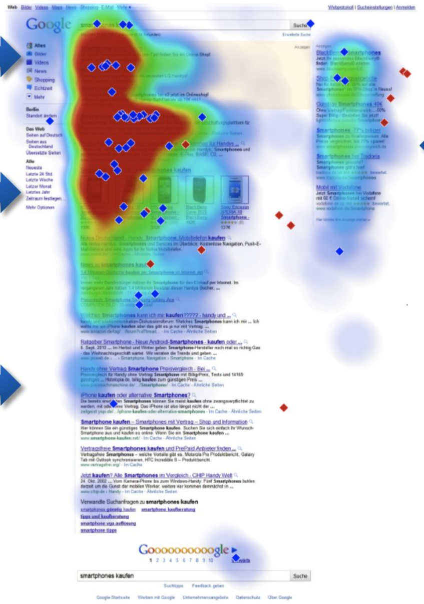
Attention of Users

Desktop search engine result page

Strong focus on the AdWords results

Rich Media elements move the attention/ fixation points down the site

Nearly no focus on the organic results below the fold



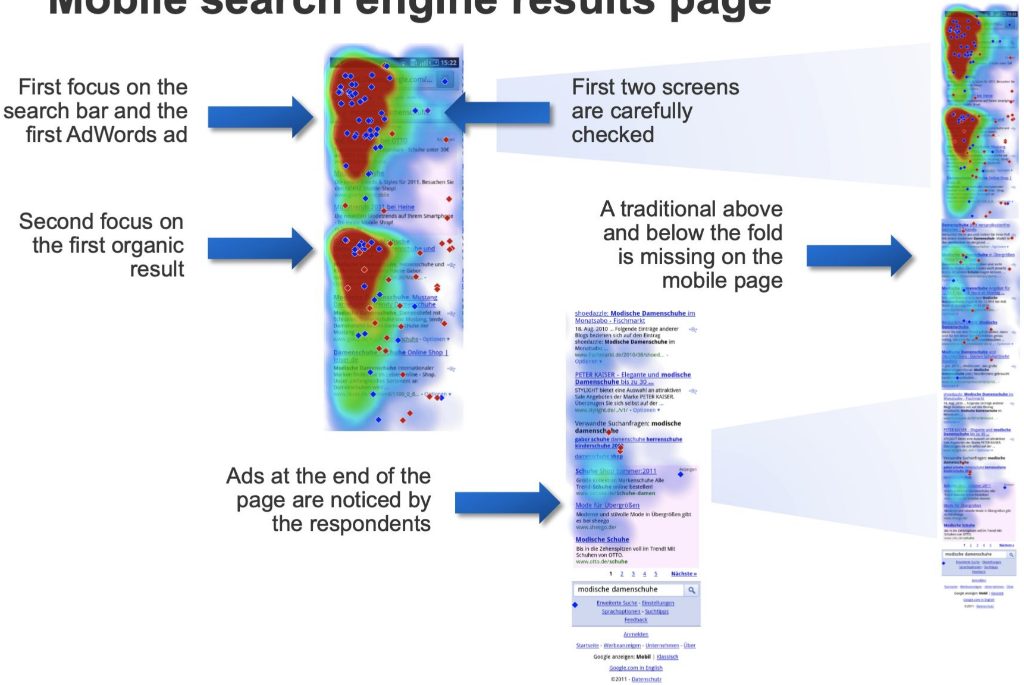
People look more intensely on the right hand side than on the organic results below the fold

thinkinsights with Google

Source: Eye Square Eye Tracking Study, 2011
Base: Resondents with contact to the stationary advertising on Google (n=38 Stationary)
Info: Aggregation over three brands

Attention of Users

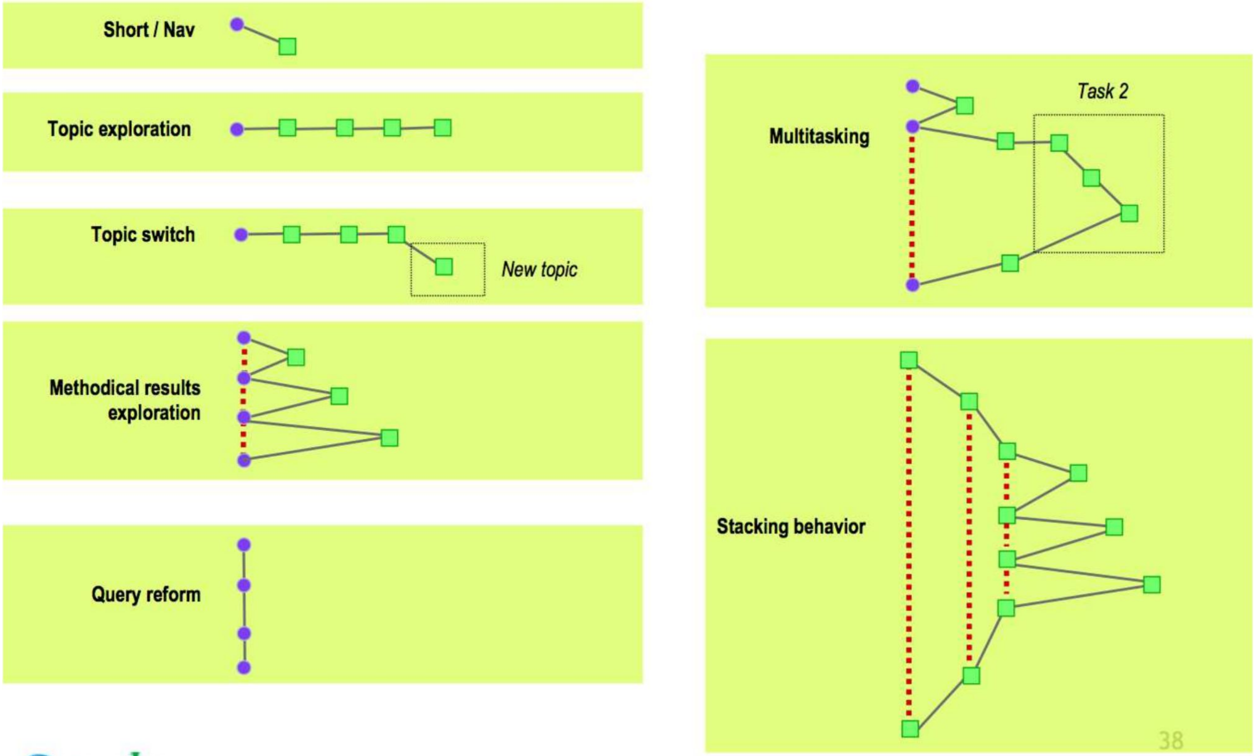
Mobile search engine results page



thinkinsights with Google

Source: Eye Square Eye Tracking Study, 2011
Base: Respondents with contact to the mobile advertising on Google (n=50 mobile)
Info: Aggregation over three brands

Users’ Behavior

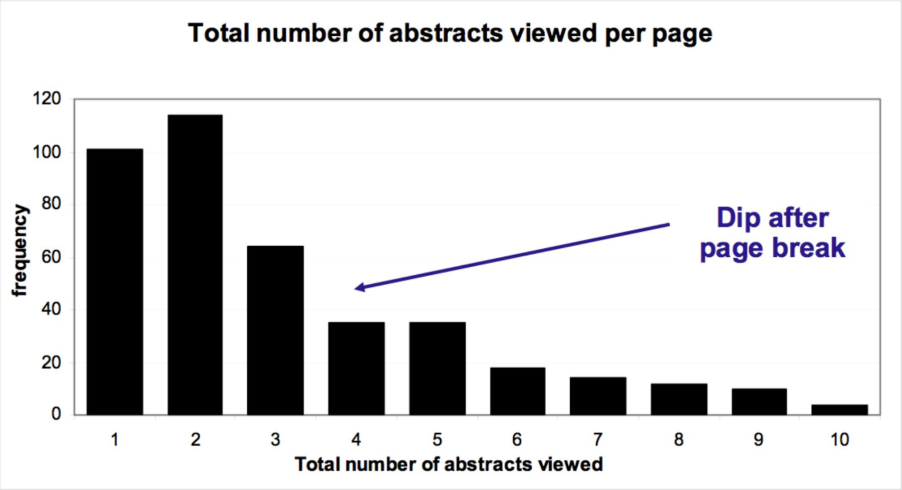


Google

SAPIENZA
UNIVERSITÀ DI ROMA

Users’ Behavior

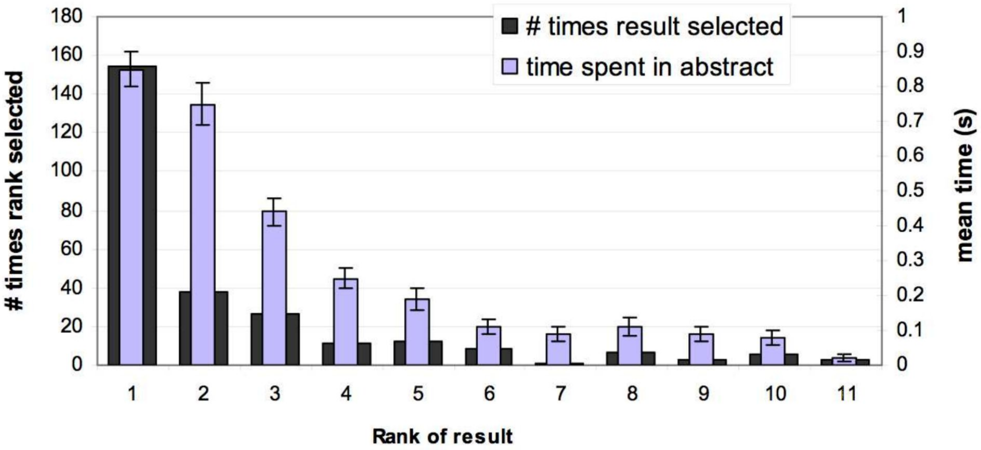
How many links do users view?



Mean: 3.07   Median/Mode: 2.00

Users' Behavior

Looking vs. Clicking



- Users view results one and two more often / thoroughly
- Users click most frequently on result one

Google

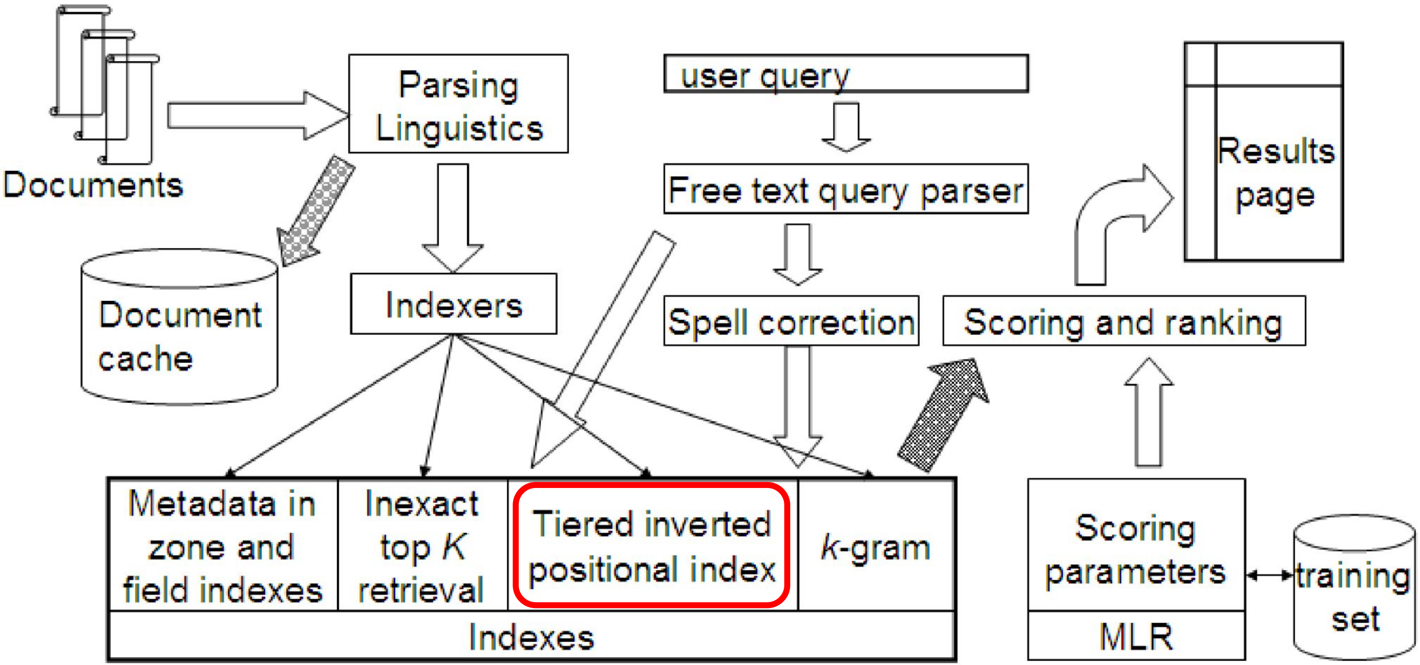
SAPIENZA UNIVERSITÀ DI ROMA

Summary

- Viewing abstracts: Users are a lot more likely to read the abstracts of the top-ranked pages (1, 2, 3, 4) than the abstracts of the lower ranked pages (7, 8, 9, 10).
- Clicking: Distribution is even more skewed for clicking
  - In 1 out of 2 cases, users click on the top-ranked page.
  - Even if the top-ranked page is not relevant, 30% of users will
- click on it.
- → Getting the ranking right is very important.
- → Getting the top-ranked page right is most important.

Real(istic) Search System

The Architecture

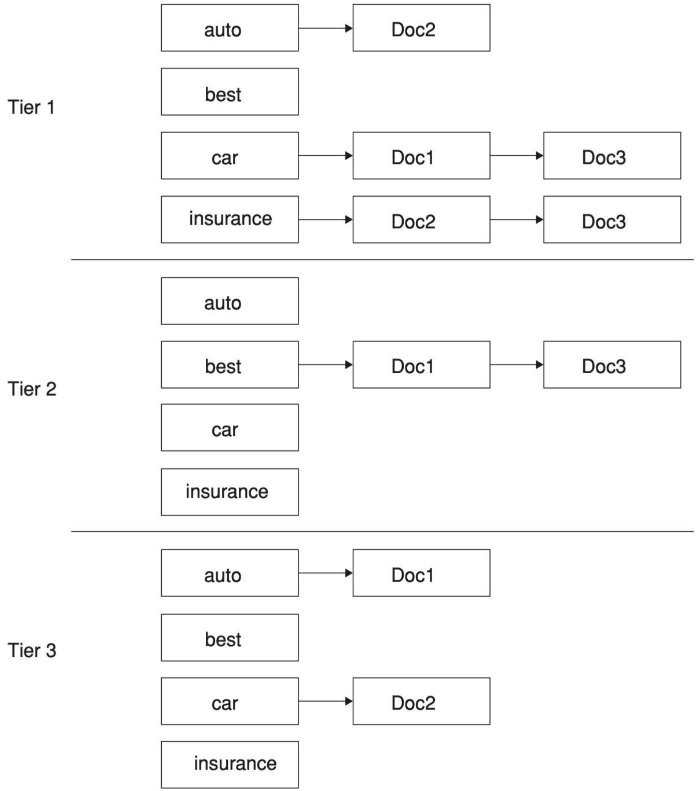


SAPIENZA
UNIVERSITÀ DI ROMA

Tiered Index

- Basic idea:
  - Create several tiers of indexes, corresponding to importance of indexing terms
  - During query processing, start with highest-tier index
  - If highest-tier index returns at least k (e.g., k = 100) results: stop and return results to user
  - If we’ve only found < k hits: repeat for next index in tier cascade
- Example: two-tier system
  - Tier 1: Index of all titles
  - Tier 2: Index of the rest of documents
  - Pages containing the search words in the title are better hits than pages containing the search words in the body of the text.

Simple Example



The Importance of Tiered Indexes

- The use of tiered indexes is believed to be one of the reasons that Google search quality was significantly higher initially (2000/01) than that of competitors.
  - (along with PageRank, use of anchor text and proximity constraints)

Efficient Scoring

Now we also need term frequencies in the index

BRUTUS	$\longrightarrow$	1,2	7,3	83,1	87,2	...
CAESAR	$\longrightarrow$	1,1	5,1	13,1	17,1	...
CALPURNIA	$\longrightarrow$	7,1	8,2	40,1	97,3	

term frequencies

Term Frequencies in the Inverted Index

- Thus: In each posting, store $tf_{t,d}$ in addition to docID d.
- As an integer frequency, not as a (log-)weighted real number. . .
  - . . . because real numbers are (more) difficult to compress.
- Overall, additional space requirements are small: a byte per posting or less

How do we compute Top-K results?

- We usually don’t need a complete ranking.
- We just need the top k for a small k (e.g., k = 100).
- If we don’t need a complete ranking, is there an efficient way of computing just the top k?
- Naive:
  - Compute scores for all N documents
  - Sort
  - Return the top k
- Not very efficient

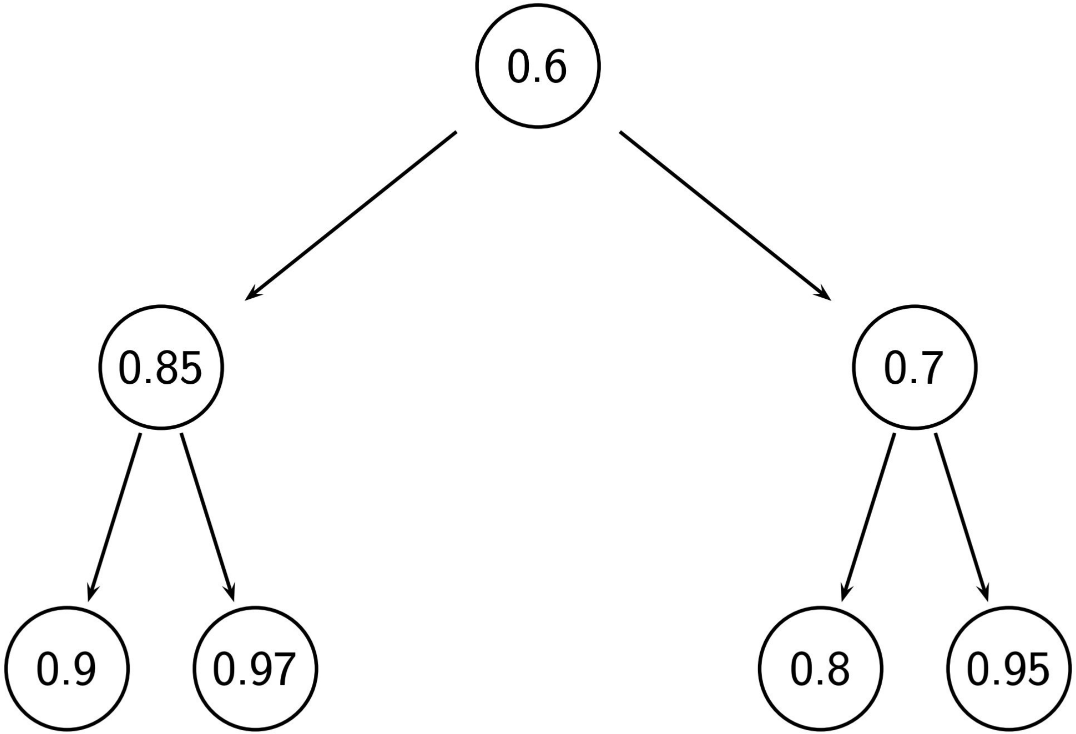
How do we compute Top-K results?

- We usually don't need a complete ranking.
- We just need the top k for a small k (e.g., k = 100).
- If we don't need a complete ranking, is there an efficient way of computing just the top k?
- Naive:
    - Compute scores for all N documents
    - Sort
    - Return the top k
- Not very efficient

Alternatively, use a Min-Heap

SAPIENZA
UNIVERSITÀ DI ROMA

What’s a Min-Heap



How do we compute Top-K results?

- Goal: Keep the top k documents seen so far
- Use a binary min heap
- To process a new document d′ with score s′:
  - Get current minimum hm of heap (O(1))
  - If $s' \leq h_m$ skip to next document
  - If $s' > h_m$ heap-delete-root (O(log k))
  - Heap-add d′/s′ (O(log k))

Even more efficient solutions?

- Ranking has time complexity O(N) where N is the number of
- documents.
- Optimizations reduce the constant factor, but they are still O(N), N > 10^{10}
- Are there sublinear algorithms?
- What we're doing in effect: solving the k-nearest neighbor (kNN) problem for the query vector (= query point).
- There are no general solutions to this problem that are sublinear.

Even more efficient solutions: Heuristics

- Idea 1: Reorder postings lists
  - Instead of ordering according to docID . . .
  - . . . order according to some measure of “expected relevance”.
- Idea 2: Heuristics to prune the search space
  - Not guaranteed to be correct . . .
  - . . . but fails rarely.
  - In practice, close to constant time.
  - For this, we’ll need the concepts of document-at-a-time processing and term-at-a-time processing.

Orderings different than DocID

- So far: postings lists have been ordered according to docID.
- Alternative: a query-independent measure of "goodness" of a page
- Example: PageRank *g(d)* of page *d*, a measure of how many "good" pages hyperlink to *d*
- Order documents in postings lists according to PageRank: *g(d1) > g(d2) > g(d3) > . . .*
- Define composite score of a document:

$$net\text{-}score(q, d) = g(d) + cos(q, d)$$

- This scheme supports early termination:
  - We do not have to process postings lists in their entirety to find top *k*.

Orderings different than DocID

- Order documents in postings lists according to PageRank:

  $g(d1) > g(d2) > g(d3) > \dots$
- Define composite score of a document:

  $\textit{net-score}(q, d) = g(d) + \cos(q, d)$
- Suppose: (i) $g \rightarrow [0, 1]$; (ii) $g(d) < 0.1$ for the document d we're currently processing; (iii) smallest top *k* score we've found so far is *1.2*

  $\rightarrow$ Then all subsequent scores will be < 1.1.
- So we've already found the top *k* and can stop processing the remainder of postings lists.

Document-at-a-Time

- Both docID-ordering and PageRank-ordering impose a consistent ordering on documents in postings lists.
- Computing cosines in this scheme is document-at-a-time.
- We complete computation of the query-document similarity score of document di before starting to compute the query-document similarity score of $d_i + 1$.
    - Alternative: term-at-a-time processing

Weight sorted posting lists

- Idea: *don't process postings that contribute little to final score*
- Order documents in postings list according to weight
- Simplest case: normalized tf-idf weight (rarely done: hard to compress)
- Documents in the top k are likely to occur early in these ordered lists.
- → Early termination while processing postings lists is unlikely to change the top k.
- But:
  - We no longer have a consistent ordering of documents in postings lists.
  - We no longer can employ document-at-a-time processing.

Term-at-a-Time Processing

- Simplest case: completely process the postings list of the first query term
- Create an accumulator for each docID you encounter
- Then completely process the postings list of the second query
- Term
  - . . . and so forth

Term-at-a-Time Processing

COSINESCORE(q)

float $Scores[N] = 0$
float $Length[N]$
for each query term t
do calculate $w_{t,q}$ and fetch postings list for t
    for each pair$(d, \text{tf}_{t,d})$ in postings list
    do $Scores[d] += w_{t,d} \times w_{t,q}$
Read the array *Length*
for each d
do $Scores[d] = Scores[d]/Length[d]$
return Top k components of $Scores[]$

The elements of the array "Scores" are called accumulators.

Accumulators

- For the web (20 billion documents), an array of accumulators *A* in memory is infeasible.
  - Thus: Only create accumulators for docs occurring in postings lists
- This is equivalent to: Do not create accumulators for docs with zero scores (i.e., docs that do not contain any of the query terms)

Accumulators

BRUTUS	→	1,2	7,3	83,1	87,2	...
CAESAR	→	1,1	5,1	13,1	17,1	...
CALPURNIA	→	7,1	8,2	40,1	97,3	

- For query: [Brutus Caesar]:
- Only need accumulators for 1, 5, 7, 13, 17, 83, 87
- Don't need accumulators for 3, 8 etc.

Enforcing Conjunctive Search

- We can enforce conjunctive search (a la Google): only consider documents (and create accumulators) if all terms occur.
- Example: just one accumulator for [Brutus Caesar] in the
- example above . . .
    - . . . because only d1 contains both words.

Ranking: Summary

- Ranking is **very expensive** in applications where we have to compute similarity scores for all documents in the collection.
- In most applications, the vast majority of documents have **similarity score 0** for a given query → lots of potential for speeding things up.
- However, there is **no fast nearest neighbor algorithm** that is guaranteed to be correct even in this scenario.
- In practice: **use heuristics** to prune search space – usually works very well.