Information Retrieval

Weike Pan

The slides are **adapted from those provided by Prof. Hinrich Schütze** at University of Munich (http://www.cis.lmu.de/~hs/teach/14s/ir/).

Chapter 7 Computing scores in a complete search system

- 7.1 Efficient scoring and ranking
- 7.2 Components of an information retrieval system
- 7.3 Vector space scoring and query operator interaction
- 7.4 References and further reading

Outline

- 7.1 Efficient scoring and ranking
- 7.2 Components of an information retrieval system
- 7.3 Vector space scoring and query operator interaction
- 7.4 References and further reading

7.1 Efficient scoring and ranking

Why is ranking so important?

- Last lecture: Problems with unranked retrieval
  - 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

7.1 Efficient scoring and ranking

Importance of ranking

- Viewing abstracts (浏览摘要): 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 (点击链接): Distribution is even more skewed (有偏的) for clicking.
- Users often 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 the most important.

7.1 Efficient scoring and ranking

Question

- Ranking is also one of the high barriers (技术壁垒) to entry for competitors to established players in the search engine market.

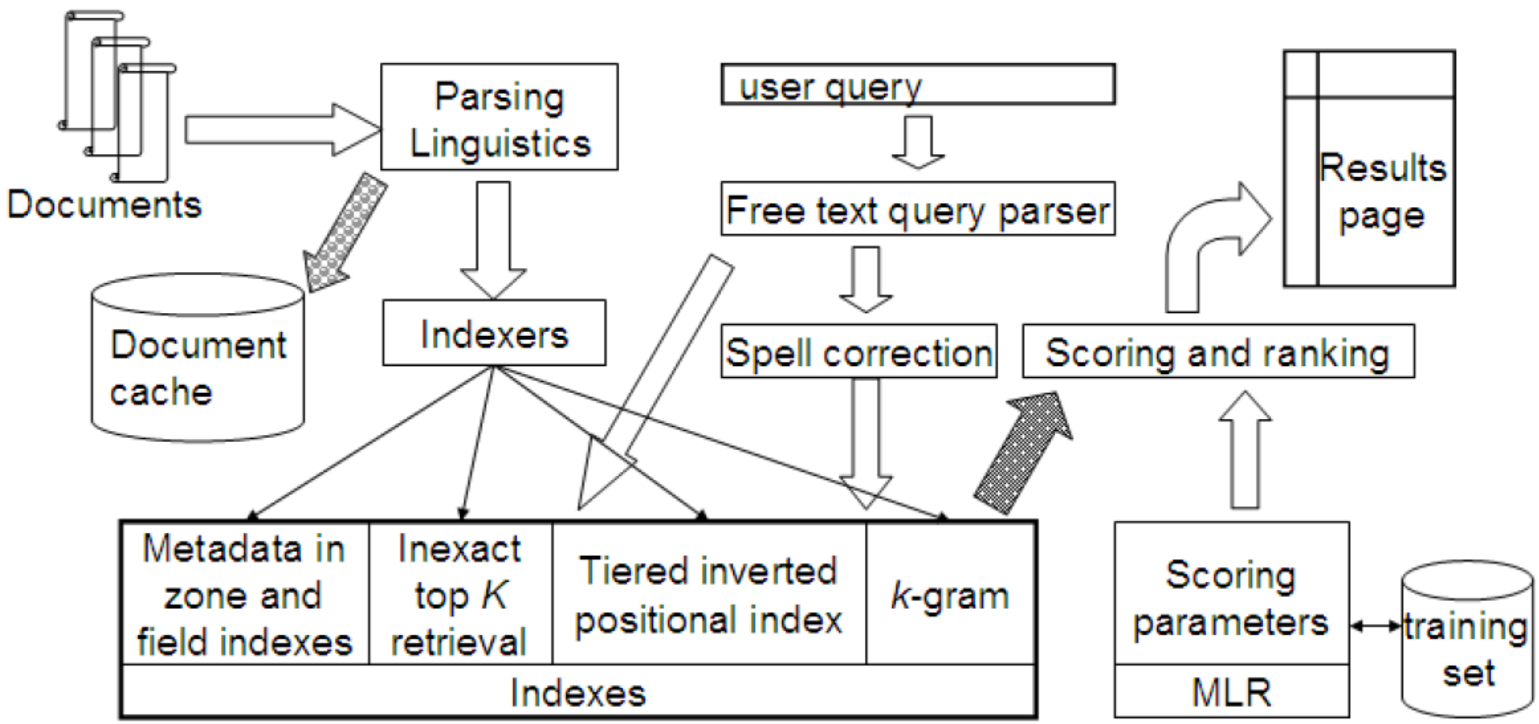
- Why?
  - Training data of query logs.

Outline

- 7.1 Efficient scoring and ranking
- 7.2 Components of an information retrieval system
- 7.3 Vector space scoring and query operator interaction
- 7.4 References and further reading

7.2 Components of an information retrieval system

Complete search system



7.2 Components of an information retrieval system

Tiered indexes (1/5)

- Basic idea:
  - Create several tiers of indexes, corresponding to the importance of the indexing terms.
  - During query processing, start with the highest-tier index.
  - If the highest-tier index returns at least k (e.g., k=100) results: stop and return results to the user.
  - If we’ve only found <k hits: repeat for the next index.

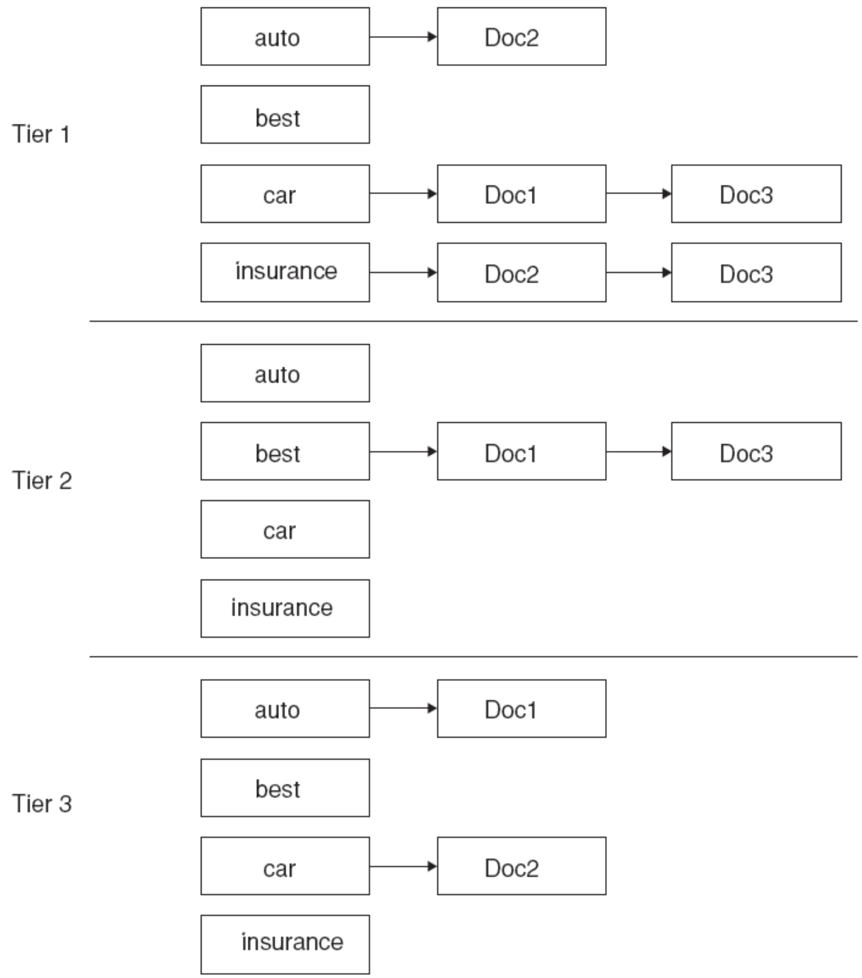
7.2 Components of an information retrieval system

Tiered indexes (2/5)

- Example: two-tier system
  - Tier 1: Index of all titles.
  - Tier 2: Index of the rest of each document.
  - Pages (i.e., documents) containing the search words in the title are better hits than pages containing the search words in the body of the text.

7.2 Components of an information retrieval system

Tiered indexes (3/5)



7.2 Components of an information retrieval system

Tiered indexes (4/5)

- **Question**:
  - Can you think of a better way of setting up a multi-tier system?
  - Which “zones” of a document should be indexed in different tiers (title, body of document, others?)?
  - What other criteria do you want to use for including a document in tier 1?

7.2 Components of an information retrieval system

Tiered indexes (5/5)

- The use of tiered indexes is believed to be one of the reasons that Google search quality was significantly higher initially (January 2000) than that of the competitors.

  (along with PageRank, use of anchor text and proximity constraints)

7.2 Components of an information retrieval system

Components we have introduced thus far

- Document preprocessing (linguistic and others)
- Positional indexes
- Tiered indexes
- Spelling correction
- k-gram indexes for wildcard queries and spelling correction
- Query processing
- Document scoring

7.2 Components of an information retrieval system

Components we haven't covered yet

- Document cache: we need this for generating snippets (=dynamic summaries, 片段、摘要)
- Zone indexes: They separate the indexes for different zones: the body of the document, all highlighted text in the document, anchor text, text in metadata fields, etc
- Machine-learned ranking functions, e.g., **learning to rank**
- Proximity ranking (e.g., rank documents in which the query terms occur in a same local window higher than documents in which the query terms occur far from each other)
- Query parser

Outline

- 7.1 Efficient scoring and ranking
- 7.2 Components of an information retrieval system
- 7.3 Vector space scoring and query operator interaction
- 7.4 References and further reading

7.3 Vector space scoring and query operator interaction

Now we also need term frequencies in the index

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	

doc ID, term frequency

7.3 Vector space scoring and query operator interaction

Term frequencies in the inverted index

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

7.3 Vector space scoring and query operator interaction

How do we compute the top k in ranking?

- 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 the N documents
  - Sort
  - Return the top k

  Not very efficient!

- Alternative: min-heap

7.3 Vector space scoring and query operator interaction

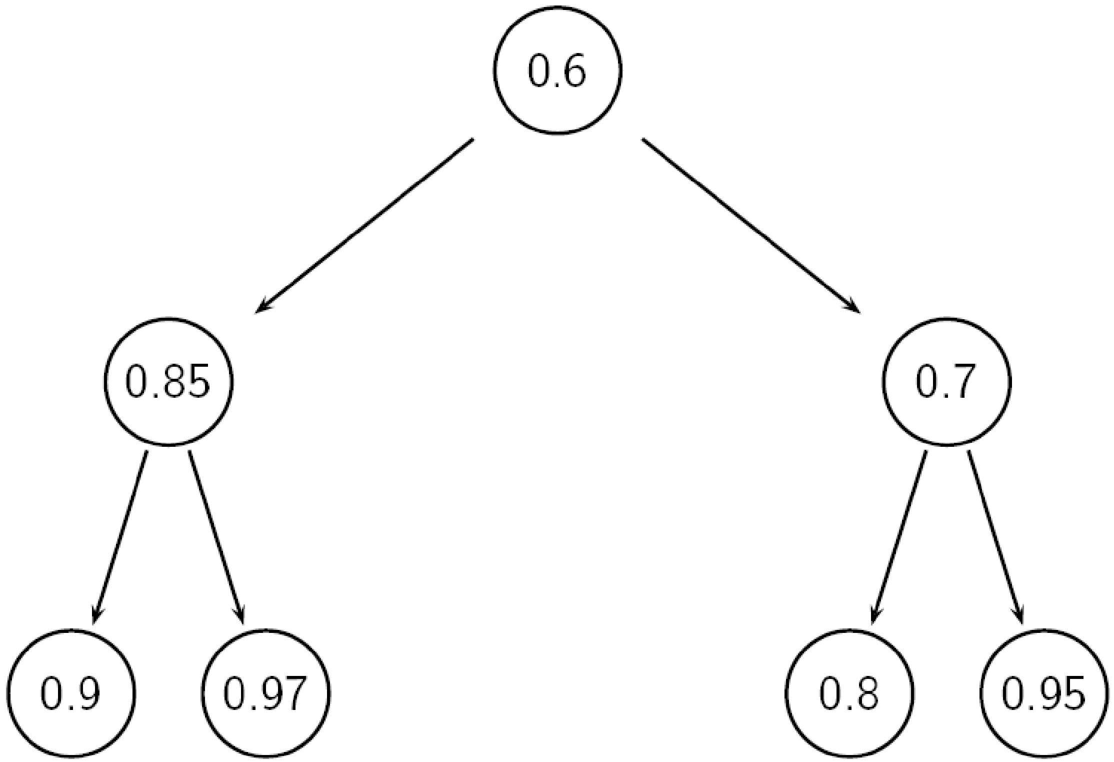
Use min-heap for selecting top k out of N

- A binary min-heap is a binary tree in which each node's value is less than (or equal to) the values of its children.

- Takes O(N log k) operations to construct (where N is the number of documents), …, then read off k winners in O(k log k) steps

7.3 Vector space scoring and query operator interaction

Binary min-heap



7.3 Vector space scoring and query operator interaction

- Selecting top k scoring documents in O(N log k)

- 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 h_m of heap, O(1)
  - If s' < h_m skip to the next document
  - If s' > h_m heap-delete-root, O(log k)

    Heap-add d', s', O(log k)

7.3 Vector space scoring and query operator interaction

Even more efficient computation of top k?

- The complexity O(N log k) includes a factor O(N), where N is the number of documents.
- Optimizations reduce the constant factor, but they are still O(N)
- Are there **sub-linear** 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 sub-linear.

7.3 Vector space scoring and query operator interaction

More efficient computation of top k: **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, it is close to constant time.
  - For this, we'll need the concepts of document-at-a-time processing and term-at-a-time processing.

7.3 Vector space scoring and query operator interaction

Heuristics for finding the top k even faster

- Document-at-a-time processing
  - We complete computation of the (query, document) similarity score of document d_i before starting to compute the (query, document) similarity score of d_{i+1}.
  - Requires a **consistent** ordering of documents in each postings list
- Term-at-a-time processing
  - We complete processing the postings list of a query term t_i before starting to process the postings list of the query term t_{i+1}.
  - Requires an **accumulator** for each document "still in the running".
- The most effective heuristics **switch back and forth** between term-at-a-time and document-at-a-time processing.

7.3 Vector space scoring and query operator interaction

Non-docID ordering of postings lists (1/2)

- So far: postings lists have been ordered according to docID.
- Alternative: a **query-independent (also, term-independent)** measure of **"goodness"** of a page
  - Example: PageRank g(d) of page d, a measure of how many "good" pages hyperlink to page d (chapter 21). Order documents in each postings list according to PageRank: g(d1) > g(d2) > g(d3) > ...

7.3 Vector space scoring and query operator interaction

Non-docID ordering of postings lists (2/2)

- Define **composite score** of a document: net-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.

- Example (例子): (i) g -> [0,1]; (ii) g(d) < 0.1 for the document d we're currently processing; (iii) the smallest top k score we've found so far is 1.2
  - Then all the subsequent scores will be **< 1.1**.
  - So we've already found the top k and can stop processing the remainder of the postings lists.

7.3 Vector space scoring and query operator interaction

Document-at-a-time processing

- Both docID-ordering and PageRank-ordering impose a **consistent** ordering on documents in postings lists.

- Computing cosine similarity score in this scheme is document-at-a-time.

- We complete computation of the (query, document) similarity score of document d_i before starting to compute the (query, document) similarity score of d_{i+1}.

7.3 Vector space scoring and query operator interaction

Weight-sorted postings lists

- Idea: don't process postings that contribute little to the final score. Order documents in a 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 because for different terms, **the ordering of two same docs may be different**. We can no longer employ document-at-a-time processing.

7.3 Vector space scoring and query operator interaction

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
  - …

7.3 Vector space scoring and query operator interaction

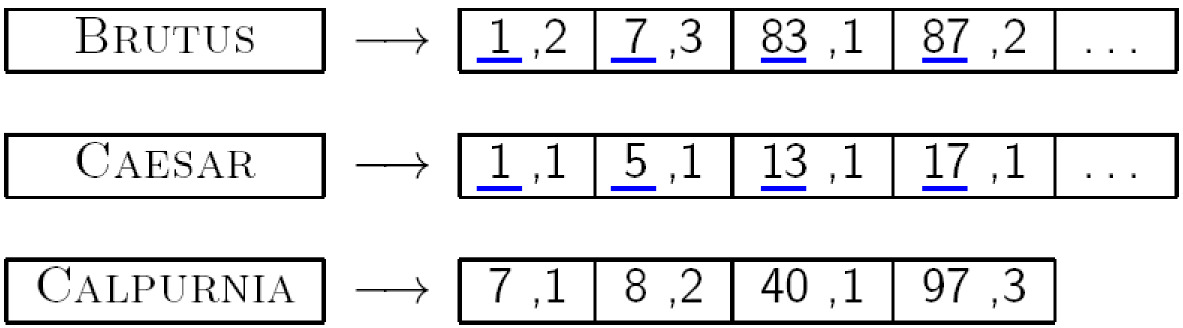
Term-at-a-time processing

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$(d, \mathrm{tf}_{t,d})$ 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.

7.3 Vector space scoring and query operator interaction

Term-at-a-time processing: Accumulators



- For query: [Brutus Caesar]
  - Only create accumulators for docs occurring in postings lists: 1, 5, 7, 13, 17, 83, 87
  - Do not create accumulators for docs with zero scores (i.e., docs that do not contain any of the query terms): 8, 40, 97

7.3 Vector space scoring and query operator interaction

Term-at-a-time processing: Enforcing conjunctive search

- We can enforce conjunctive search (e.g., 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.

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	

7.3 Vector space scoring and query operator interaction

Implementation of ranking: Summary

- Ranking is very expensive in applications where we have to compute similarity scores for **all** the documents in the collection.
- In most applications, the vast majority of documents have a similarity score 0 for a given query. Hence, there is 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.

Summary

- 7.1 Efficient scoring and ranking
- 7.2 Components of an information retrieval system
- 7.3 Vector space scoring and query operator interaction
- 7.4 References and further reading