Introduction to Information Retrieval

http://informationretrieval.org

IIR 7: Scores in a Complete Search System

Mihai Surdeanu

(Based on slides by Hinrich Schütze at informationretrieval.org)

Fall 2015

Overview

Outline

Term frequency weight

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

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

idf weight

- 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

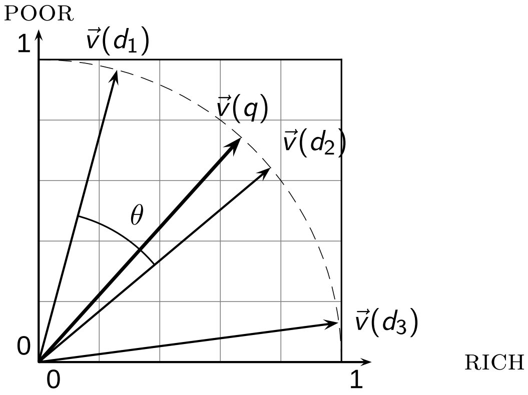
- The tf-idf weight of a term is the product of its tf weight and its idf weight.
- $$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

Cosine similarity between query and document

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

- 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.
- $|\vec{q}|$ and $|\vec{d}|$ are the lengths of $\vec{q}$ and $\vec{d}$.
- $\vec{q}/|\vec{q}|$ and $\vec{d}/|\vec{d}|$ are length-1 vectors (= normalized).

Cosine similarity illustrated



tf-idf example: lnc.ltn

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	tf-wght	n’lized	
auto	0	0	5000	2.3	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0	0	0	0	0
car	1	1	10000	2.0	2.0	1	1	1	0.52	1.04
insurance	1	1	1000	3.0	3.0	2	1.3	1.3	0.68	2.04

Key to columns: tf-raw: raw (unweighted) term frequency, tf-wght: logarithmically weighted term frequency, df: document frequency, idf: inverse document frequency, weight: the final weight of the term in the query or document, n’lized: document weights after cosine normalization, product: the product of final query weight and final document weight

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

Final similarity score between query and document: $\sum_i w_{qi} \cdot w_{di} = 0 + 0 + 1.04 + 2.04 = 3.08$

Take-away today

- The importance of ranking: User studies at Google
- The complete search system
- Implementation of ranking

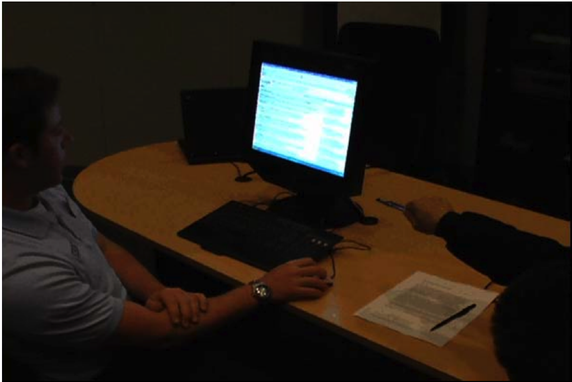
Outline

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
  - $\rightarrow$ Ranking is important because it effectively reduces a large set of results to a very small one.
- Next: More data on "users only look at a few results"
- Actually, in the vast majority of cases they only examine 1, 2, or 3 results.

Empirical investigation of the effect of ranking

- The following slides are from Dan Russell's JCDL talk
- Dan Russell was the "Über Tech Lead for Search Quality & User Happiness" at Google.
- How can we measure how important ranking is?
- Observe what searchers do when they are searching in a controlled setting
  - Videotape them
  - Ask them to "think aloud"
  - Interview them
  - Eye-track them
  - Time them
  - Record and count their clicks



So.. Did you notice the FTD official site?

To be honest, I didn't even look at that.

At first I saw "from $20" and $20 is what I was looking for.

To be honest, 1800-flowers is what I'm familiar with and why I went there next even though I kind of assumed they wouldn't have $20 flowers

And you knew they were expensive?

I knew they were expensive but I thought "hey, maybe they've got some flowers for under $20 here…"

But you didn't notice the FTD?

No I didn't, actually… that's really funny.

Interview video

Rapidly scanning the results

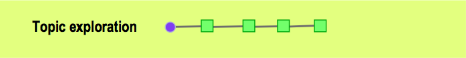
Note scan pattern:

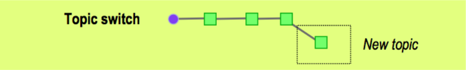
Page 3:	Result 1
	Result 2
	Result 3
	Result 4
	Result 3
	Result 2
	Result 4
	Result 5
	Result 6 <click>

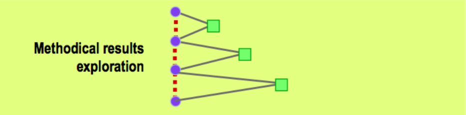
Q: Why do this?

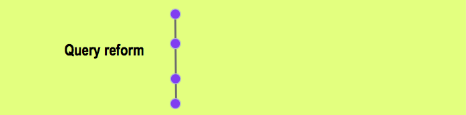
A: What's learned later influences judgment of earlier content.

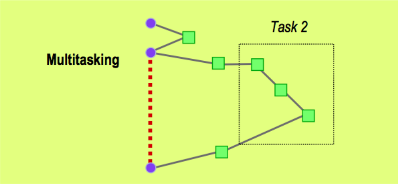
Kinds of behaviors we see in the data

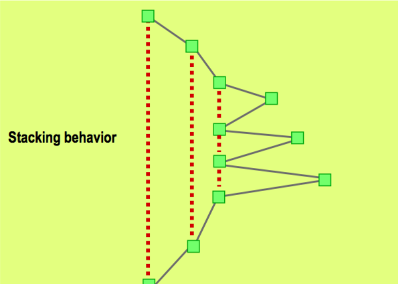




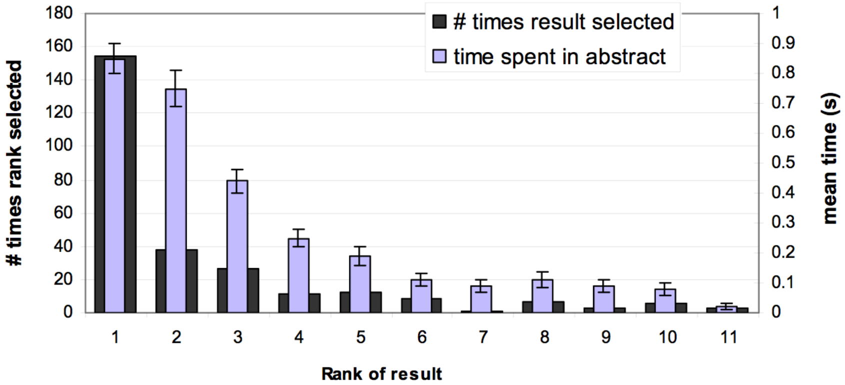








Looking vs. Clicking

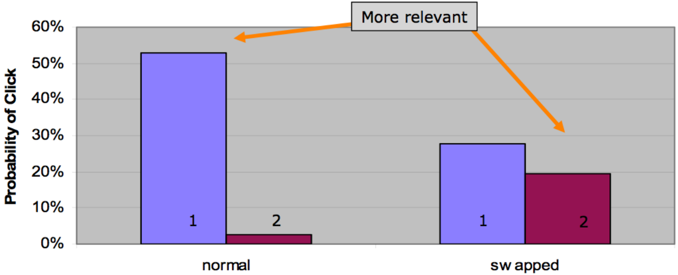


- Users view results one and two more often / thoroughly
- Users click most frequently on result one

Google

Presentation bias – reversed results

- Order of presentation influences where users look **AND** where they click



Importance of ranking: 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.
- $\rightarrow$ Getting the ranking right is very important.
- $\rightarrow$ Getting the top-ranked page right is most important.

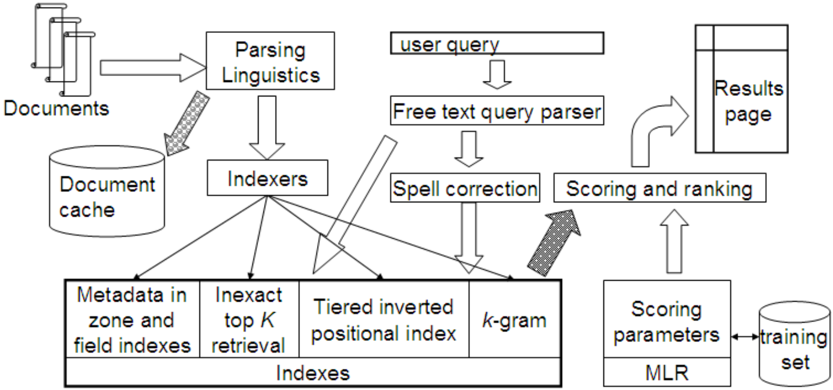
Exercise

- Ranking is also one of the high barriers to entry for competitors to established players in the search engine market.
- Why?

Outline

1. Recap
2. Why rank?
3. The complete search system
4. Implementation of ranking

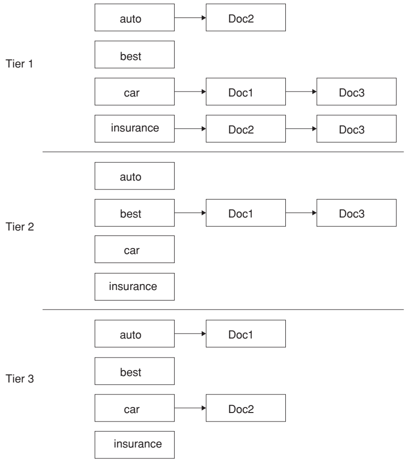
Complete search system



Tiered indexes

- 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.

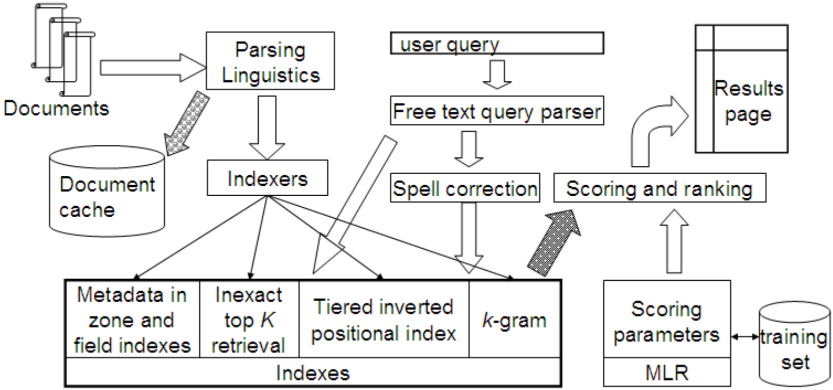
Tiered index: by term frequency



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)
    - What are these?

Complete search system



Components we have introduced thus far

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

Components we haven't covered yet

- Document cache: we need this for generating snippets (= dynamic summaries)
- 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. Why?
- Proximity ranking (e.g., rank documents in which the query terms occur in the same local window higher than documents in which the query terms occur far from each other)
- The two issues above best covered using machine-learned ranking functions!
  - But Google doesn't do it. Why?
- Query parser (we've seen an example for Lucene!)

Components we haven't covered yet: Query parser

- IR systems often guess what the user intended.
- The two-term query *London tower* (without quotes) may be interpreted as the phrase query *"London tower"*.
- The query *100 Madison Avenue, New York* may be interpreted as a request for a map.
- How do we "parse" the query and translate it into a formal specification containing phrase operators, proximity operators, indexes to search etc.? Need query syntax language!
  - We have seen Lucene's, but actual search engines do a lot of natural language processing as well, e.g., to recognize addresses, definitional questions, etc.

Exercise: Interactions with vector space retrieval

- How do we combine phrase retrieval with vector space retrieval?
- We do not want to compute document frequency / idf for every possible phrase. Why?
- How do we combine Boolean retrieval with vector space retrieval?
- How do we combine wild cards with vector space retrieval?

Exercise: Better tiered system

- Design criteria for tiered system
  - Each tier should be an order of magnitude smaller than the next tier.
  - The top 100 hits for most queries should be in tier 1, the top 100 hits for most of the remaining queries in tier 2 etc.
  - We need a simple test for "can I stop at this tier or do I have to go to the next one?"
    - There is no advantage to tiering if we have to hit most tiers for most queries anyway.
- Consider a two-tier system where the first tier indexes titles and the second tier everything.
- Question: Can you think of a better way of setting up a multitier system? Which "zones" of a document should be indexed in the different tiers (title, body of document, others?)? What criterion do you want to use for including a document in tier 1?

Outline

Now we also need term frequencies in the index

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	

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

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

We also need positions. Not shown here.

Term frequencies in the inverted index

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

Computing cosine: just like Boolean intersection!

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, $tf_{t,d}$) 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 $Scores[]$

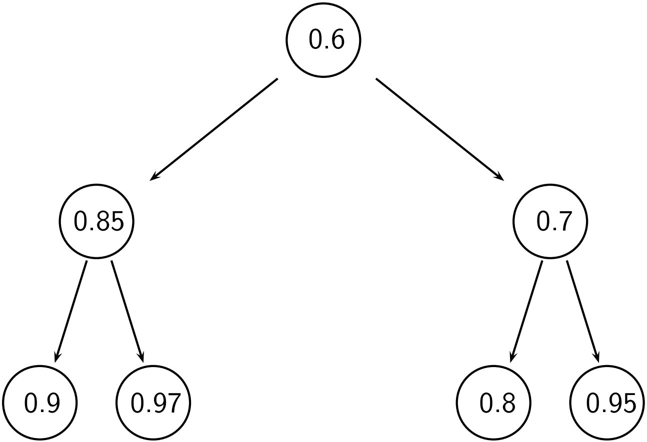
How do we compute the top k in ranking?

- 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 N documents
  - Sort
  - Return the top k
- Not very efficient
- Alternative: min heap

Use min heap for selecting top k ouf of N

- A binary min heap is a binary tree in which each node's value is less than the values of its children.
- It is a complete tree: all levels are completely filled, except possibly the last one. If the last level is not complete, the leaves are filled from left to right.
- 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

Binary min heap: insert 0.75



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 $s' \leq h_m$ skip to next document
  - If $s' > h_m$ heap-delete-root ($O(\log k)$)
  - Heap-add d'/s' ($O(\log k)$)

Even more efficient computation of top k?

- 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?

Even more efficient computation of top k?

- 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.

More efficient computation of top k: Three ideas (heuristics)

- Document-at-a-time processing
- Term-at-a-time processing
- Cluster pruning

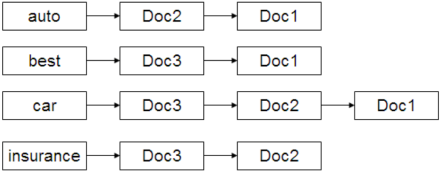
Non-docID ordering of postings lists

- 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 (chapter 21)
- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \dots$
- Define composite score of a document:

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

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

Non-docID ordering of postings lists (2)



► **Figure 7.2** A static quality-ordered index. In this example we assume that Doc1, Doc2 and Doc3 respectively have static quality scores $g(1) = 0.25, g(2) = 0.5, g(3) = 1$.

Postings no longer sorted by doId, but by g! Does it matter?

Non-docID ordering of postings lists (3)

- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \ldots$
- Define composite score of a document:

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

Non-docID ordering of postings lists (3)

- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \dots$
- Define composite score of a document:

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

- 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

Non-docID ordering of postings lists (3)

- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \ldots$
- Define composite score of a document:

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

- Suppose: (i) $g \to [0, 1]$; (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
- Then all subsequent scores will be < 1.1.

Non-docID ordering of postings lists (3)

- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \dots$
- Define composite score of a document:

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

- Suppose: (i) $g \to [0, 1]$; (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
- 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.

Non-docID ordering of postings lists (3)

- Order documents in postings lists according to PageRank: $g(d_1) > g(d_2) > g(d_3) > \ldots$
- Define composite score of a document:

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

- 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
- 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.
- Questions?

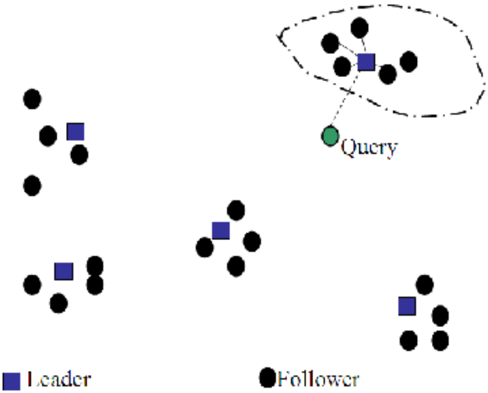
This was document-at-a-time processing

- 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.
- This means we need to be careful to store intermediate results for all documents seen!
- Alternative: term-at-a-time processing

Term-at-a-time processing

- Idea 1 (sort postings):
  - Order the documents d in the postings list of term t by decreasing order of $tf_{t,d}$.
  - When traversing the postings list for a query term t, we stop either after a fixed number of documents r have been seen, or after the value of $tf_{t,d}$ has dropped below a threshold.
- Idea 2 (sort query):
  - Sort terms in query q in descending order of idf.
  - Stop after getting to terms with low idf values.

Cluster pruning



Implementation of 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 $\rightarrow$ 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.

Take-away today

- The importance of ranking: User studies at Google
- The complete search system
- Implementation of ranking