Informatics 225
Computer Science 221

Information Retrieval

Lecture 22

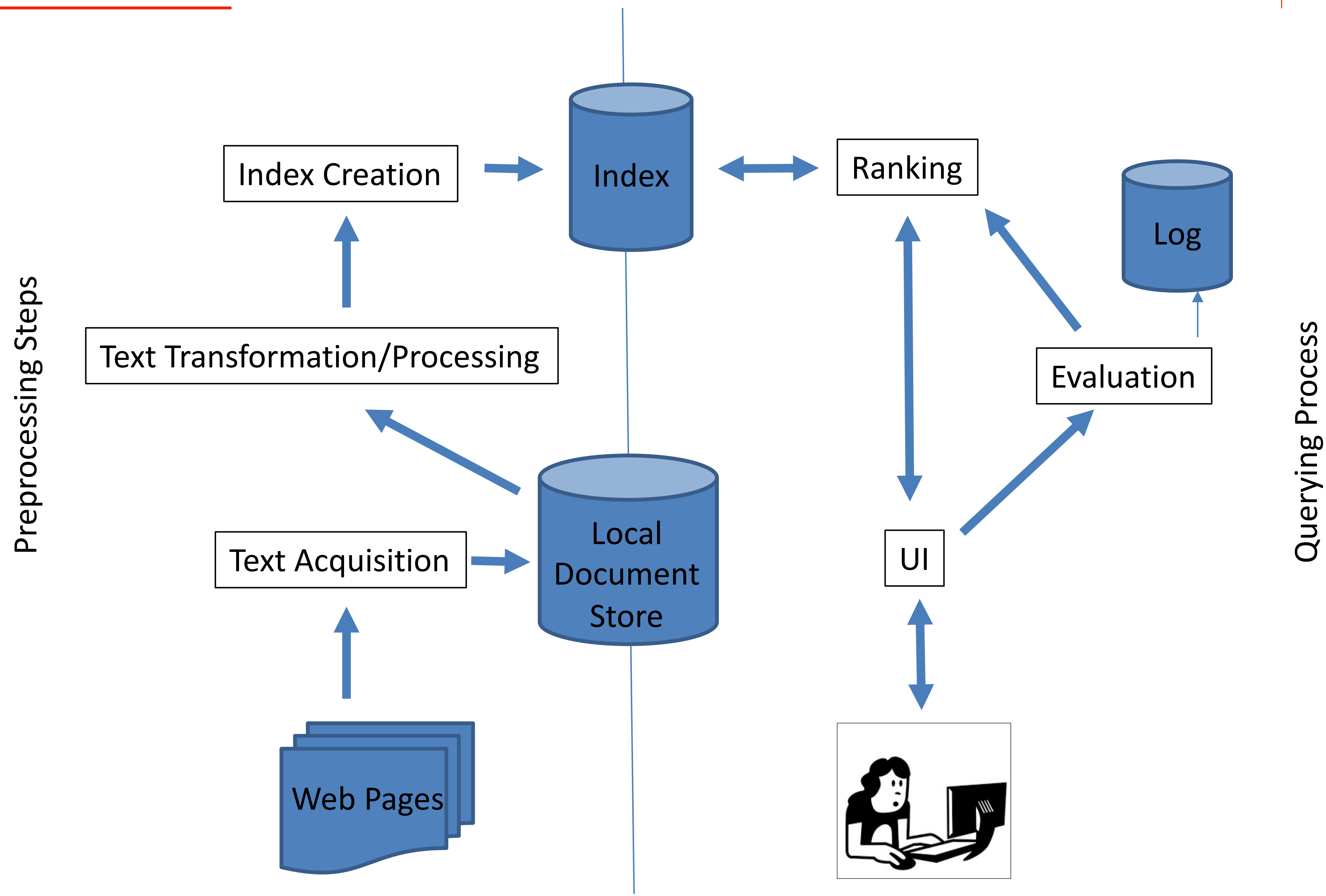
These course materials borrow, with permission, from those of Prof. Cristina Videira Lopes, Addison Wesley 2008, Chris Manning, Pandu Nayak, Hinrich Schütze, Heike Adel, Sascha Rothe, Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie. Powerpoint theme by Prof. André van der Hoek.

Scoring and result assembly

Information Retrieval

These course materials borrow, with permission, from those of Prof. Cristina Videira Lopes, and additional materials from Addison Wesley 2008, Chris Manning, Pandu Nayak, Hinrich Schütze, Heike Adel and Sascha Rothe

Architecture



Recap: tf-idf weighting

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

$$\mathrm{w}_{t,d} = (1 + \log_{10} \mathrm{tf}_{t,d}) \times \log_{10}(N / \mathrm{df}_t)$$

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Recap: Queries as vectors

- **<u>Key idea 1:</u>** Do the same for queries: represent them as vectors in the space

- **<u>Key idea 2:</u>** Rank documents according to their proximity to the query in this space

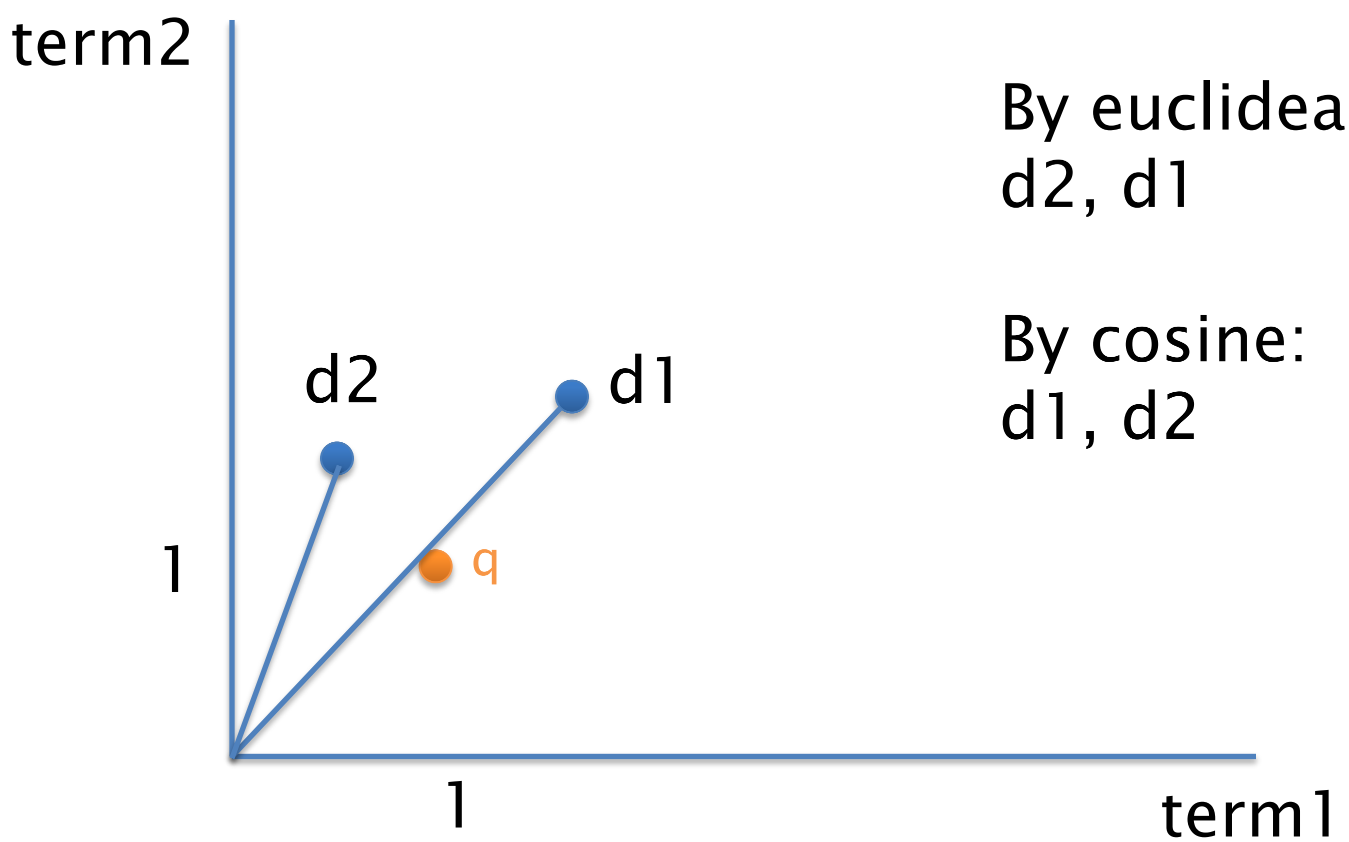
- proximity = similarity of vectors
- proximity ≈ inverse of distance

Recap: cosine(query,document)

$$\cos(\vec{q},\vec{d}) = \frac{\overbrace{\vec{q} \bullet \vec{d}}^{\text{Dot product}}}{|\vec{q}||\vec{d}|} = \overbrace{\frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|}}^{\text{Unit vectors}} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

Simple tf-idf vs. cosine scoring



By euclidean tf-idf:
d2, d1

By cosine:
d1, d2

Now…

- Speeding up vector space ranking

- Starting to put together a complete search system
  - Will require us to consider miscellaneous topics and **heuristics**

Computing cosine scores

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 $Scores[]$

Efficient cosine ranking

- Find the K docs in the collection "nearest" to the query $\Rightarrow$ K largest query-doc cosines.

Efficient cosine ranking

- Find the *K* docs in the collection "nearest" to the query $\Rightarrow$ *K* largest query-doc cosines.

- Efficient ranking:
  – Computing a single cosine efficiently.

Efficient cosine ranking

- Find the K docs in the collection "nearest" to the query $\Rightarrow$ K largest query-doc cosines.

- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the K largest cosine values efficiently.

Efficient cosine ranking

- Find the K docs in the collection "nearest" to the query $\Rightarrow$ K largest query-doc cosines.

- Efficient ranking:
  - Computing a single cosine efficiently.
  - Choosing the K largest cosine values efficiently.
    - **Can we do this without computing all *N* cosines?**

Efficient cosine ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector

Efficient cosine ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector

- In general, we do not know how to do this efficiently for high-dimensional spaces

Efficient cosine ranking

- What we’re doing in effect: solving the *K*-nearest neighbor problem for a query vector

- In general, we do not know how to do this efficiently for high-dimensional spaces

- But we don’t need to solve for the very high dimensional spaces in the context of websearch! Query vectors are highly sparse!

Efficient cosine ranking

- What we're doing in effect: solving the *K*-nearest neighbor problem for a query vector

- In general, we do not know how to do this efficiently for high-dimensional spaces

- But we don't need to solve for the very high dimensional spaces in the context of websearch! Query vectors are highly sparse!
  - The problem is solvable for short queries, and standard indexes support this well

Special case – unweighted queries

- No weighting on query terms
  - Assume each query term occurs only once
- Then for ranking, don't need to normalize query vector

Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - And not to totally order all docs in the collection

Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - And not to totally order all docs in the collection

- Can we pick off docs with *K* highest cosines?

Computing the *K* largest cosines: selection vs. sorting

- Typically we want to retrieve the top *K* docs (in the cosine ranking for the query)
  - And not to totally order all docs in the collection
- Can we pick off docs with *K* highest cosines?
- Let *J* = number of docs with nonzero cosines
  - We seek the *K* best of these *J*

Use heap for selecting top *K*

- Binary tree in which each node's value > the values of children
- Takes *2J* operations to construct, then each of *K* "winners" read off in 2log *J* steps.
- For *J*=1M, *K*=100, this is about ~10% of the cost of sorting!

Bottlenecks

- Primary computational bottleneck in scoring: cosine computation

Bottlenecks

- Primary computational bottleneck in scoring: <u>cosine computation</u>

- Can we avoid all this computation?

Bottlenecks

- Primary computational bottleneck in scoring: cosine computation

- Can we avoid all this computation?

- Yes, but may sometimes get it wrong

Bottlenecks

- Primary computational bottleneck in scoring: cosine computation
- Can we avoid all this computation?
- Yes, but may sometimes get it wrong
  - a doc ***not*** in the top *K* may creep into the list of *K* output docs

Bottlenecks

- Primary computational bottleneck in scoring: cosine computation

- Can we avoid all this computation?

- Yes, but may sometimes get it wrong
  - a doc ***not*** in the top *K* may creep into the list of *K* output docs
  - **Is this such a bad thing?**

Cosine similarity…

- User has a task and a query formulation

Cosine similarity…

- User has a task and a query formulation

- Cosine matches docs to query

Cosine similarity is only a proxy!

- User has a task and a query formulation
- Cosine matches docs to query
- Thus cosine is anyway a proxy for user happiness

Cosine similarity is only a proxy!

- User has a task and a query formulation
- Cosine matches docs to query
- Thus cosine is anyway a proxy for user happiness
- **If we get a list of *K* docs "close" to the top *K* by cosine measure, should be ok**

Generic approach

- Find a set A of *contenders*, with $K < |A| << N$

Generic approach

- Find a set *A* of *contenders*, with $K < |A| << N$
  - *A* does not necessarily contain the top *K*,
  - but has many docs from among the top *K*
  - Return the top *K* docs in *A*

Generic approach

- Find a set *A* of *contenders*, with $K < |A| << N$
  - *A* does not necessarily contain the top *K*,
  - but has many docs from among the top *K*
  - Return the top *K* docs in *A*

- Think of *A* as <u>pruning</u> non-contenders

Generic approach

- Find a set A of *contenders*, with $K < |A| << N$
  - A does not necessarily contain the top K,
  - but has many docs from among the top K
  - Return the top K docs in A
- Think of A as <u>pruning</u> non-contenders
- The same approach is also used for other (non-cosine) scoring functions
- Will look at a few schemes following this approach

Index elimination

- Basic algorithm for cosine computation only considers docs containing at least one query term

Index elimination

- Basic algorithm for cosine computation only considers docs containing at least one query term

- Take this heuristic further:
  – Only consider high-idf query terms
  – Only consider docs containing many query terms

Index elimination: High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*

Index elimination: High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: ***in*** and ***the*** contribute little to the scores and so don’t alter rank-ordering much

Index elimination: High-idf query terms only

- For a query such as *catcher in the rye*
- Only accumulate scores from *catcher* and *rye*
- Intuition: ***in*** and ***the*** contribute little to the scores and so <u>don't alter rank-ordering much</u>
- **Benefit**:
  - Postings of low-idf terms have many docs → these (many!!!) docs get eliminated from set *A* of contenders

Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list

Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list

- For multi-term queries, only compute scores for docs containing several of the query terms (perhaps all terms!)

Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms (perhaps all terms!)
  - Say, at least 3 out of 4

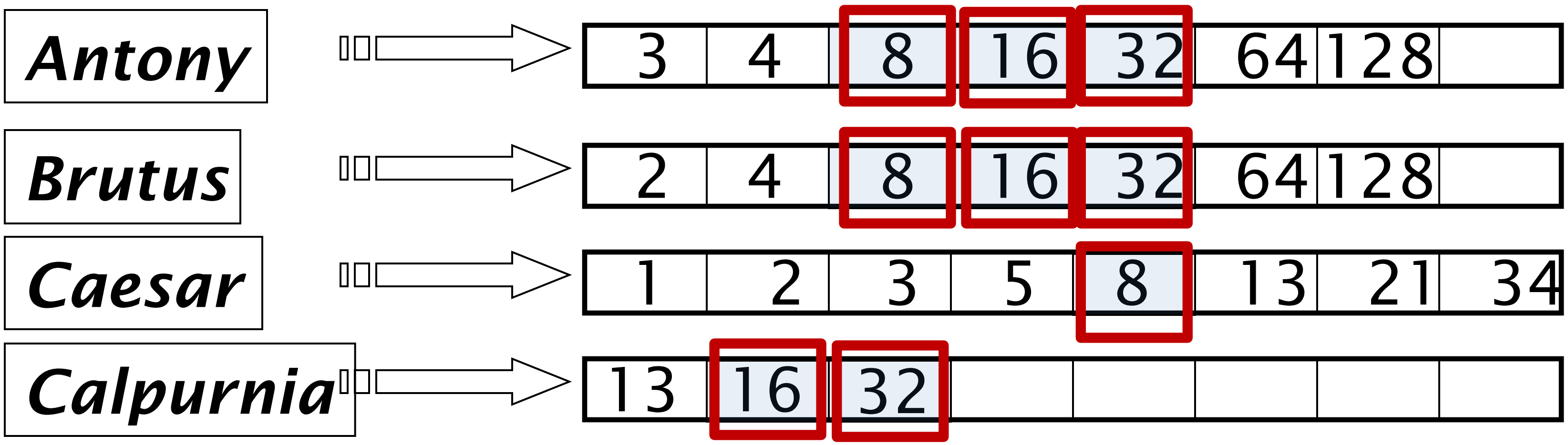
Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms (perhaps all terms!)
  - Say, at least 3 out of 4
  - Imposes a "soft conjunction" on queries seen on web search engines (early Google)

Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms (perhaps all terms!)
  - Say, at least 3 out of 4
  - Imposes a "soft conjunction" on queries seen on web search engines (early Google)
  - May end up with fewer than K candidates if too restrictive in the number of query terms. Add another heuristic on top: *you can iterate!*

Example 3 of 4 query terms



Scores only computed for docs 8, 16 and 32.

If you only want to show 2 docs to the user, it is enough.
If you need more, you iterate and now compute scores for 2 query terms also.

Index elimination: Docs containing many query terms

- Any doc with at least one query term is a candidate for the top *K* output list
- For multi-term queries, only compute scores for docs containing several of the query terms (perhaps all terms!)
  - Say, at least 3 out of 4
  - Imposes a “soft conjunction” on queries seen on web search engines (early Google)
  - May end up with fewer than K candidates if too restrictive in the number of query terms. Add another heuristic on top: *you can iterate!*
- Easy to implement in postings traversal