Chapter 7
Computing scores in a complete search system
Content

- Speeding up vector space ranking
- Putting together a complete search system
Efficiency bottleneck

- Top-k retrieval: we want to find the $K$ docs in the collection “nearest” to the query $\Rightarrow K$ largest query-doc cosines.
- 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 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
- Thus, it’s acceptable to do inexact top k document retrieval
Inexact top K: generic approach

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

- The same approach is also used for other (non-cosine) scoring functions

- Will look at several schemes following this approach
Index elimination

- Only consider high-idf query terms

- Only consider docs containing many query terms
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 don’t alter rank-ordering much
- Benefit:
  - Postings of low-idf terms have many docs → these (many) docs get eliminated from A
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
  - Say, at least 3 out of 4
  - Imposes a “soft conjunction” on queries seen on web search engines (early Google)
- Easy to implement in postings traversal
3 of 4 query terms

Antony

| 3 | 4 | 8 | 16 | 32 | 64 | 128 |

Brutus

| 2 | 4 | 8 | 16 | 32 | 64 | 128 |

Caesar

| 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |

Calpurnia

| 13 | 16 | 32 |

Scores only computed for 8, 16 and 32.
Champion lists

- Precompute for each dictionary term $t$, the $r$ docs of highest weight in $t$’s postings
  - Call this the champion list for $t$
  - (aka fancy list or top docs for $t$)

- **Note:** postings are sorted by docID, a common order

- Note that $r$ has to be chosen at index time
  - $r$ not necessarily the same for different terms

- At query time, only compute scores for docs in the champion list of some query term
  - Pick the $K$ top-scoring docs from amongst these
We want top-ranking documents to be both relevant and authoritative.

Relevance is being modeled by cosine scores.

Authority is typically a query-independent property of a document.

Examples of authority signals:
- Wikipedia among websites
- Articles in certain newspapers
- A paper with many citations
- Many diggs, Y!buzzes or del.icio.us marks
- (Pagerank)
Modeling authority

- Assign to each document a *query-independent* quality score in $[0,1]$ to each document $d$
  - Denote this by $g(d)$
- Thus, a quantity like the number of citations is scaled into $[0,1]$
Net score

- Consider a simple total score combining cosine relevance and authority
- \( \text{net-score}(q,d) = g(d) + \text{cosine}(q,d) \)
  - Can use some other linear combination than an equal weighting
- Now we seek the top \( K \) docs by net score
Top $K$ by net score – idea 1

- Order all postings by $g(d)$
  - Key: this is a common ordering for all postings
- Thus, can concurrently traverse query terms’ postings for
  - Postings intersection
  - Cosine score computation
- Under $g(d)$-ordering, top-scoring docs likely to appear early in postings traversal
  - In time-bound applications (say, we have to return whatever search results we can in 50 ms), this allows us to stop postings traversal early
Top $K$ by net score – idea 2

- Can combine champion lists with $g(d)$-ordering
- Maintain for each term a champion list of the $r$ docs with highest $g(d) + \text{tf-idf}_{td}$
- Seek top-$K$ results from only the docs in these champion lists
- **Note:** postings are sorted by $g(d)$, a common order
For each term, we maintain two postings lists called high and low

Think of high as the champion list

When traversing postings on a query, only traverse high lists first

If we get more than $K$ docs, select the top $K$ and stop

Else proceed to get docs from the low lists

Can be used even for simple cosine scores, without global quality $g(d)$

A means for segmenting index into two tiers

Tiered indexes (later)
Impact-ordered postings

- We only want to compute scores for docs for which $\text{wf}_{t,d}$ is high enough.
- We sort each postings list by $\text{tf}_{t,d}$ or $\text{wf}_{t,d}$.
- Now: not all postings in a common order!
  - If common order (docID, g(d)), supports concurrent traversal of all query terms’ posting lists. Computing scores in this manner is referred to as “document-at-a-time scoring”
  - Otherwise, “term-at-a-time”

- How do we compute scores in order to pick off top $K$?
  - Two ideas follow
1. Early termination

- When traversing \( t \)'s postings, stop early after either
  - a fixed number of \( r \) docs
  - \( \text{wf}_{t,d} \) drops below some threshold
- Take the union of the resulting sets of docs
  - One from the postings of each query term
- Compute only the scores for docs in this union
2. idf-ordered terms

- When considering the postings of query terms
- Look at them in order of decreasing idf
  - High idf terms likely to contribute most to score
- As we update score contribution from each query term
  - Stop if doc scores relatively unchanged
- Can apply to cosine or some other net scores
Cluster pruning: preprocessing

- Pick $\sqrt{N}$ docs at random: call these leaders
  - Why random?
  - Fast; leaders reflect data distribution

- For every other doc, pre-compute nearest leader
  - Docs attached to a leader: its followers;
  - Likely: each leader has $\sim \sqrt{N}$ followers.
Cluster pruning: query processing

- Process a query as follows:
  - Given query Q, find its nearest leader L.
  - Seek \( K \) nearest docs from among L’s followers.
Visualization

- **Leader**
- **Follower**

Query
Content

- Speeding up vector space ranking
- Putting together a complete search system
  - Components of an IR system
Thus far, a doc has been a sequence of terms.

In fact documents have multiple parts, some with special semantics:
- Author, Date of publication, Language, Format, Title

These constitute the metadata about a document.

This metadata would generally include fields such as the date of creation and the format of the document, as well the author and possibly the title of the document.

The possible values of a field should be thought of as finite - for instance, the set of all dates of authorship.
We sometimes wish to search by these metadata

- E.g., find docs authored by William Shakespeare in the year 1601, containing *alas poor Yorick*

**Parametric (or field) index:** there is one parametric index for each field (say, date of creation)

- Parametric search typically treated as conjunction
  - doc *must* be authored by Shakespeare
Zone indexes

- Zones are similar to fields, except the contents of a zone can be arbitrary free text, whereas a field may take on a relatively small set of values. For instance, document titles and abstracts are generally treated as zones.
  - Title, Abstract, References …

- Build inverted indexes on zones as well to permit querying, e.g.,
  - find documents with merchant in the title and the phrase gentle rain in the body
Example zone indexes

Encode zones in dictionary vs. postings.

william.abstract → 11 → 121 → 1441 → 1729

william.title → 2 → 4 → 8 → 16

william.author → 2 → 3 → 5 → 8

william → 2.author,2.title → 3.author → 4.title → 5.author
Tiered indexes

- Break postings up into a hierarchy of lists
  - Most important
  - ...
  - Least important
- Can be done by $g(d)$ or another measure
- Inverted index thus broken up into tiers of decreasing importance
- At query time use top tier unless it fails to yield $K$ docs
  - If so drop to lower tiers
Example tiered index

Tier 1
- auto → Doc2
- best
- car → Doc1 → Doc3
- insurance → Doc2 → Doc3

Tier 2
- auto
- best → Doc1 → Doc3
- car
- insurance

Tier 3
- auto → Doc1
- best
- car → Doc2
- insurance
Query term proximity

- Free text queries: just a set of terms typed into the query box – common on the web
- Users prefer docs in which query terms occur within close proximity of each other
- Let $w$ be the smallest window in a doc containing all query terms, e.g.,
  - For the query *strained mercy* the smallest window in the doc *The quality of mercy is not strained* is 4 (words)
- Would like scoring function to take this into account – how?
Query parsers

- Free text query from user may in fact spawn one or more queries to the indexes, e.g. query *rising interest rates*
  - Run the query as a phrase query
  - If <K docs contain the phrase *rising interest rates*, run the two phrase queries *rising interest* and *interest rates*
  - If we still have <K docs, run the vector space query *rising interest rates*
  - Rank matching docs by vector space scoring
- This sequence is issued by a query parser
Aggregate scores

- We’ve seen that score functions can combine cosine, static quality, proximity, etc.
- How do we know the best combination?
- Some applications – expert-tuned
- Increasingly common: machine-learned
Putting it all together