Chapter 7: Computing scores in a complete search system

- Speedups for cosine scoring
- How to build a complete search engine?
- Vector space model and query operators

7.1 Efficient scoring and ranking

- For the purpose of ranking, we are interested in *relative* scores of the documents in the collection.
- Hence it suffices to compute the cosine similarity from each document unit vector $\vec{v}(d)$ to $\vec{V}(q)$, where all non-zero components of the query vector are set to 1, rather than to the unit vector $\vec{v}(q)$.
  - For any two documents d_1, d_2, $\vec{V}(q) \cdot \vec{v}(d_1) > \vec{V}(q) \cdot \vec{v}(d_2) \Leftrightarrow \vec{v}(q) \cdot \vec{v}(d_1) > \vec{v}(q) \cdot \vec{v}(d_2)$.

7.1.1 Inexact top K document retrieval

- Instead of retrieving precisely the top K, let's come up with K documents that are *likely* to be among the K highest scoring documents.
  - Dramatically lowering the computing costs, without materially altering the user's *perceived* relevance of the top K results.
  - Cosine similarity is a proxy anyway.
- The principal computing cost comes from calculating similarities between the query and *a large number of documents*.
- So we need to get many documents out of consideration without calculating their scores, using the heuristics with the two-step scheme:
  1. Find a set A of documents that are contenders, where $K < |A| \ll N$. A does not necessarily contain the K top-scoring documents for the query, but is likely to have many documents with scores near those of the top K.
  2. Return the K top-scoring documents in A.
- Many of these heuristics will require many parameter tunings.
- These are for free text queries and not for Boolean or phrase queries.

7.1.2 Index elimination

- For a multi-term query q, we already consider only the documents containing at least one of the query terms. We could use more heuristics.

1. Only consider documents containing terms with *high enough idf*: The postings lists of low idf terms are generally long. Basically we now consider them as stop words, they end up not contributing anything to the scoring.
   - Cutoff threshold can be adapted in a *query-dependent manner*.
2. Only consider documents containing *many* (sometimes all) of the query terms.
   - We might end up with fewer than K candidates.

7.1.3 Champion lists

- *Champion list*: For each term t in the dictionary, precompute the set of r documents with the highest weights for t.
  - r should be chosen in advance.
- Then make the set A the *union* of the champion lists for each of the terms comprising q, and restrict cosine computation to only the documents in A.
  - Hence r should be fairly larger than K.
  - One issue is that r would be set during the index construction, while K is application dependent.
- No need to set the same value of r for all terms: we might set it higher for rarer terms.

7.1.4 Static quality scores and ordering

- **Static quality score**: A measure of quality $g(d)$ for each document d that is query-*independent* and thus *static*.
  - A number between 0 and 1
- Then the *net score* for a document d is some combination of $g(d)$ together with the query-dependent score.
- Using these static quality scores, we could create postings lists by *decreasing* value of $g(d)$ and perform the postings intersection.
  - Note that what we needed for postings intersection was a *single common ordering* between all postings.
- *Global champion list*: Extension of a regular champion list. Maintain the list with the highest values for $g(d) + \text{tf-idf}_{t,d}$.
- Maintaining *two* postings lists consisting of *disjoint* sets of documents
  - *High*: the list containing the m documents with the highest tf values for t.
  - *Low*: the list containing *all other* documents containing t.

- Then we can first try scanning only through high lists of all query terms. We go through low lists only if we don't get K documents.

7.1.5 Impact ordering

- A technique for when the postings are not all ordered by a common ordering, thereby precluding a concurrent traversal (which was possible by traversing all of the query terms' postings lists and scoring each document as we encounter it)
- *Term*-at-a-time scoring instead of document-at-a-time scoring.
- Idea: Order the documents in the postings list of term t by *decreasing* order of $tf_{t,\mathbf{d}}$.
  - When going through each postings list for a term t, stop after considering a fixed number of documents or after the value of $tf_{t,d}$ has dropped below a threshold.
  - When accumulating scores, we consider each query terms in decreasing order of *idf*, so that the query terms likely to contribute *the most* to the final scores are considered first.
  - When we process a query, we can also determine whether to continue processing the remaining query terms after looking at the changes from the previous query term processed.
- *Impact ordering*: Ordering by something other than term frequencies

7.1.6 Cluster pruning

- Consider only documents *in a small number of clusters* as candidates
  1. *Leaders*: Pick $\sqrt{N}$ documents at random from the collection.
  2. *Followers*: For each document that is not a leader, we compute its nearest leader.
     - The expected number of followers for each leader is $\approx N/\sqrt{N} = \sqrt{N}$.
  3. Given a query q, find the leader L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ leaders.
  4. The candidate set A consists of L together with its followers.
- Using randomly chosen leaders for clustering is fast and more likely to reflect *the distribution of the document vectors in the vector space.
- Variations: additional parameters of positive integers b_1 and b_2.
  - Attach each follower to b_1 closest leaders instead of a single leader
  - At query time, we consider the b_2 leaders closest to the query q.
  - In the standard version above, $b_1 = b_2 = 1$.
  - Raising b_1 or b_2 higher increases the likelihood of finding K documents that are more likely to be in the set of true top-scoring K documents.

7.2 Components of an information retrieval system

- A rudimentary search system that retrieves and scores documents.
- We do not restrict ourselves to vector space retrievals.

7.2.1 Tiered indexes

- *Tiered indexes*: If we fail to get K results from tier 1, query processing falls back to tier 2, and so on.
  - For example: Tier 1 have a *tf* threshold of 20, and 2 have 10, and so on.

7.2.2 Query-term proximity

- Especially for free text queries on the web, users prefer to find documents in which *most or all* of the query terms *appear close to each other.*
  - Because this is the evidence that the document has text *focused on their query intent.*
- Consider a query with 2 or more query terms, $t_1, t_2, \cdots, t_k$.
- Let ω be the *width* of the smallest window in a document d that contains *all the query terms*, measured by the number of words in the window.
- The smaller that ω is, the better that d matches the query.
  - In case where the document does not contain all of the query terms, we can set ω to be some enormous number.
  - Also consider variants in which only words that are *not stop words* are considered in computing ω.
- Such proximity-weighted scoring functions are a departure from pure cosine similarity and closer to the *soft conjunctive* semantics that web search engines use.
- How should we set ω?
  - Hard coding
  - Machine learning

7.2.3 Designing parsing and scoring functions

- Common search interfaces tend to mask query operators and encourage free text queries
  - Then how should we combine query features?
  - The answer depends on
    * The user population
    * The query population
    * The collection of documents
- Typically, a *query parser* is used to translate the user-specified keywords into a query with various operators
  - Sometimes, this execution can entail *multiple queries* against the underlying indexes
  1. Run the user-generated query string as a phrase query, rank them with vector space scoring, treating the query as a vector containing all query terms.
  2. If the step 1 contained too few documents, run phrase queries of length one term shorter.
  3. If we didn't get enough documents in previous steps, then run the vector space query consisting of individual query terms.
- Scores must combine contributions from vector space scoring, static quality, proximity weighting and potentially other factors
  - Particularly since a document may appear in the lists from multiple steps.
  - Need an aggregate scoring function that accumulates evidence of a document's relevance from multiple sources.
  - The answer depends on the setting (enterprise search vs. web search)

7.2.4 Putting it together

- Brief review of how the various pieces fit together into an overall system.

7.3 Vector space scoring and query operator interaction

- How the vector space scoring model relates to the query operators
  - in terms of the *expressiveness* of queries
  - in terms of the index that supports the evaluation
- Classic interpretation of free text queries: At least one of the query terms be present in any retrieved document
- More recently: A set of terms carries the semantics of a *conjunctive* query that only retrieves documents containing *all or most* query terms
- *Boolean retrieval*: While a vector space index can used to answer Boolean queries, the reverse is not true as a Boolean index does not carry any weight information.
- *Wildcard queries*: If a search engine allows a user to specify a wildcard operator as part of a free text query, we may interpret the wildcard component query as spawning multiple terms in the vector space.
  - All of those terms would be added to the query vector.
  - A document containing multiple of the terms is likely to be scored higher than another containing fewer of them.
  - The exact score ordering will depend on the relative weights of each term in matching documents
- *Phrase queries*: An index built for vector space retrieval cannot be used directly for phrase retrieval
  - Even if we model biwords as terms, the weights on different axes wouldn't be independent
  - Notions such as idf would have to be extended to such biwords
  - We could use vector space retrieval to identify documents heavy in individual query terms, but with no way of prescribing that they occur consecutively.
    * Phrase retrieval can do exactly that, but without any indication of the relative frequency or weight in this phrase.