Information Retrieval and Organisation

Dell Zhang

Birkbeck, University of London
Computing Scores in a Complete Search System
Inexact Top-\(k\) Retrieval

- We now consider schemes which produce \(k\) documents that are likely to be among the \(k\) highest scoring documents
  - We hope to dramatically lower the cost of computing the top-\(k\) documents
  - Obviously, we don’t want to alter the user’s perceived relevance of the top-\(k\) results significantly
- May not be such a bad thing as it sounds like
  - Cosine similarity is also only a proxy for the user’s perceived relevance
Inexact Top-$k$ Retrieval

- We’ll now look at some ideas designed to eliminate a large number of documents without computing their cosine scores.
- These heuristics follow a two-step scheme:
  1. Find a set $A$ of documents that are contenders, where $k < |A| \ll N$.
    - $A$ does not necessarily contain all the $k$ top-scoring documents for the query, but there should be a large overlap.
  2. Return the $k$ top-scoring documents in $A$. 
Index Elimination

- We could only consider the terms whose idf exceeds a certain threshold
  - Low idf means that terms are not very relevant
  - These terms tend to have very long postings lists
- We could only consider the documents that contain many (or all) query terms
  - Only compute cosine values for these documents
  - The danger is that we could end up with $|A| < k$ (we’ll come back to this in a moment)
Champion Lists

- Pre-compute, for each term $t$ in the dictionary, the set of the $r$ documents with the highest tf-values for $t$. We call this set of $r$ documents the champion list for term $t$ (sometimes also called fancy list or top docs).

- We create $A$ by combining the champion lists of all terms in query $q$.

- Determining the parameter $r$ is crucial
  - As $r$ is determined when constructing the index, we might not know $k$ then
  - So we might choose an $r$ that is too small (ending up with $|A| < k$ again)
Static Quality Scores

- In many search engines, a query-independent measure of quality is available
- The scores calculated based on such measures are called *static quality scores*
  - For example, the number of favourable reviews of news stories
- The matching-score is computed by combining the static quality $g(d)$ of a document $d$ with other query-dependent scores
  - A simple way to do this would be to add $g(d)$ to the cosine measure
- Such static quality scores can be used to build champion lists based on $g(d)$
Impact Ordering

- The algorithm \textsc{COSINESCORE} in the last chapter applied a document-at-a-time processing
  - That means, for each \( d, \text{tf}_{t,d} \) pair we calculated the cosine measure
  - We have to accumulate the score for each document while the algorithm is running

- This is very inefficient:
  - We have to store scores for millions or even billions of documents
  - Most of those documents will never make it into the top-\( k \)
Impact Ordering

- Naturally, we only want to compute cosine measures for serious contenders (the set $A$)
- So we allocate space for computing $|A|$ scores
- How do we make sure that we process the most important documents first?
Impact Ordering

- Up to now we have implicitly assumed that postings lists are ordered by docIDs
- However, if we add term frequencies (or other scores such as $g(d)$) and want to do inexact top-$k$ retrieval, other orders might be better
- Let’s assume that we have postings lists with term frequency values (each entry consists of (docID, tf-value))
  - e.g., information, 3: $\langle(1, 3), (2, 1), (5, 2)\rangle$
- We could order the postings lists in decreasing order of tf-values:
  - e.g., information, 3: $\langle(1, 3), (5, 2), (2, 1)\rangle$
Impact Ordering

- We access the postings lists of all the terms contained in the query
- Then we process the items in the lists in decreasing tf-value order
  - Heuristic: documents in the top-\(k\) are likely to occur early in these ordered lists
- We can also extend this scheme with idf-values, i.e. multiply each tf-value with the idf-value of the term before deciding on the order
- The first \(|A|\) documents encountered get their total scores computed
Impact Ordering

Here’s an example for three postings lists (and simplified tf-idf-values):

- information, \( \text{idf}=1; 3: \langle (1,3), (5,2), (2,1) \rangle \); 
- line, \( \text{idf}=3; 2: \langle (2,6), (1,2) \rangle \); 
- computer, \( \text{idf}=2; 5: \langle (3,7), (5,4), (2,3), (1,2), (4,1) \rangle \); 

Start with document 2, term line:

- \( 3 \times 6 = 18 \); largest tf-idf value

Continue with document 3, term computer:

- \( 2 \times 7 = 14 \); second-larges tf-idf value

and so on . . .
Storing TF values

- Storing the tf-values for all documents will take up considerable space
  - The first problem we face is: how do we store the tf-values efficiently?
  - As it turns out, unary coding is quite good at this.

<table>
<thead>
<tr>
<th>method</th>
<th>Bible</th>
<th>GNUBib</th>
<th>Comact</th>
<th>TREC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>1.27</td>
<td>1.16</td>
<td>1.74</td>
<td>2.49</td>
</tr>
<tr>
<td>Gamma</td>
<td>1.38</td>
<td>1.23</td>
<td>1.88</td>
<td>2.13</td>
</tr>
</tbody>
</table>
Storing TF values

- However, when sorting by tf-values we have problems with compressing docIDs (as gap encoding relies on sorted docIDs)
  - For example, the list
    \[
    \langle 5 : (1, 2), (2, 2), (3, 5), (4, 1), (5, 2) \rangle
    \]
    would be sorted like this
    \[
    \langle 5 : (3, 5), (1, 2), (2, 2), (5, 2), (4, 1) \rangle
    \]
- Solution: organize items in “tf-blocks”
  \[
  (tf, k : d_1, \ldots, d_k),
  \]
  where \( k \) is the number of documents for a certain tf-value and the \( d_i \)'s are sorted docIDs
  - So for the above example, we would get:
    \[
    \langle 5 : (5, 1 : 3), (2, 3 : 1, 2, 5), (1, 1 : 4) \rangle
    \]
  - Needs slightly more memory than a docID-sorted list, but still efficient
Cluster Pruning

- In *cluster pruning*, we have a preprocessing step during which we cluster the document vectors
  - Pick $\sqrt{N}$ documents at random from the collection, we call these *leaders*.
  - For each document that is not a leader, we compute its nearest leader.
  - We refer to documents that are not leaders as *followers*.
  - The expected number of followers for each leader is roughly $N/\sqrt{N} = \sqrt{N}$

- We’ll talk about more advanced text clustering techniques later in the module
Cluster Pruning

▶ At query time, we only compute cosine measures for a small number of documents
  ▶ Given a query $q$, find the leader $L$ closest to $q$ (this entails computing cosine similarities from $q$ to each of the $\sqrt{N}$ leaders)
  ▶ The candidate set $A$ consists of $L$ together with its followers (this entails computing cosine similarities from $q$ to each of the $\sqrt{N}$ followers)
Cluster Pruning

- Leader
- Follower

Query
Tiered Indexes

- Create several tiers of indexes, corresponding to importance of indexing terms
- During query processing, start with the highest-tier index
- If we get $\geq k$ hits: stop and return the results to user
- If we get $< k$ hits: repeat for the next index in tier cascade
Tiered Indexes

- **Example: two-tier system**
  - Tier 1: Index of all titles
  - Tier 2: Index of the rest of documents
  - As pages containing the search words in the title are usually better hits than pages containing the search words in the body of the text.

- **Could be expanded to three-tier system**
  - Tier 1: Index of all titles
  - Tier 2: Index of all abstracts
  - Tier 3: Index of the rest of documents
Tiered Indexes
Putting It All Together
What Have We Covered So 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
- Term-at-a-time processing
What Is Yet To Come?

- Document cache
  - e.g., for generating snippets (dynamic summaries)
- Zone indexes
  - 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
- 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
- Query Parser
  - see next slide
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” or even “Tower of London”.
- 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.?
Summary

- Different variants for computing scores
- How to compute scores efficiently (inexact top-$k$ retrieval)
- How a complete retrieval system looks like