SEARCH ENGINES: INFORMATION RETRIEVAL IN PRACTICE SELECTED EXERCISE SOLUTIONS

W. BRUCE CROFT, DONALD METZLER, AND TREVOR STROHMAN

Exercise 2.2. A *more-like-this* query occurs when the user can click on a particular document in the result list and tell the search engine to find documents that are similar to this one. Describe which low-level components are used to answer this type of query and the sequence in which they are used.

A more-like-this query is answered by first retrieving the document of interest from the document data store. Depending on how the document is represented in the document data store, it may have to be parsed, stopped, and stemmed. The end result of this process is a set of terms which are then weighted according to some term weighting scheme, such as $tf.idf$. A query is then constructed from this weighted set of terms using the underlying search engine's query language. This query is then used to score all of the documents in the collection. The highest scoring documents are then displayed to the user.

Exercise 2.3. Document filtering is an application that stores a large number of queries or user profiles and compares these profiles to every incoming document on a feed. Documents that are sufficiently similar to the profile are forwarded to that person via email or some other mechanism. Describe the architecture of a filtering engine and how it may differ from a search engine.

Document filtering systems and search engines are very similar. In search engines, incoming queries are matched against a collection of documents. In document filtering, incoming documents are matched against a collection of profiles. Therefore, a document filtering system can use an architecture similar to a search engine. The primary differences between the two architectures is the type of items being indexed, the nature of the queries, and how users interact with the systems.

Document filtering systems need to acquire (via user interaction), transform, and index user profiles, instead of documents. In addition, the document filtering system needs to automatically transform incoming documents into queries. The queries distilled from the incoming documents are then used to find relevant profiles.

Exercise 3.2. Suppose you have a network connection that can transfer 10MB per second. If each web page is 10K and requires 500 milliseconds to transfer, how many threads does your web crawler need to fully utilize the network connection? If your crawler needs to wait 10 seconds between requests to the same web server, what is the minimum number of distinct web servers the system needs to contact each minute to keep the network connection fully utilized?

Since each 10K page requires 500ms to transfer (including latency), we can compute the rate at which a page can be downloaded using a single thread as follows:

$$\frac{10 \text{ KB}}{\text{page}} \cdot \frac{\text{page}}{500 \text{ ms}} \cdot \frac{1000 \text{ ms}}{\text{s}} = \frac{20\text{KB}}{\text{s}}$$

At this rate, the crawler would need 512 threads to fully utilize the 10MB per second network connection.

Each of the 512 threads can download 2 pages per second, or 20 pages per 10 seconds. Thus, each thread needs to contact, at the minimum, 20 distinct web servers every 10 second cycle. Since each thread can contact the same set of 20 web servers every 10 seconds, the crawler would need to contact a total minimum of $512 \cdot 20 = 10\text{K}$ distinct web servers per minute.

Exercise 3.3. What is the advantage of using HEAD requests instead of GET requests during crawling? When would a crawler use a GET request instead of a HEAD request?

Crawlers can use HEAD requests when trying to determine if a web page has changed since the last time it was crawled. Since HEAD requests only return a very small payload (see Figure 3.5), such requests can significantly reduce the amount of data transferred compared to always using GET requests, thereby increasing the efficiency of the crawler.

A crawler would use a GET request the first time it crawls a page or if a GET request has indicated that the page has been updated since it was last crawled.

Exercise 3.8. Suppose that, in an effort to crawl web pages faster, you set up two crawling machines with different starting seed URLs. Is this an effective strategy for distributed crawling? Why or why not?

This is not a very effective strategy for distributed crawling, because the two machines are unable to share information with each other. Despite the fact that the two machines are seeded with different URLs, this would lead to a great deal of duplicated effort, because the two machines would eventually crawl many of the same URLs. This strategy could be made more effective by sharing the URL request queue between the two machines, thereby eliminating the duplicated effort.

Exercise 4.10. Figure 4.11 shows an algorithm for computing PageRank. Prove that the entries of the vector I sum to 1 every time the algorithm enters the loop on line 9.

Before the first iteration of the algorithm every entry of the vector I equals $\frac{1}{|P|}$ (lines 6-8). Therefore, the sum of the entries is 1 since there are $|P|$ entries in I. During each iteration, page p's entry in I is calculated as follows:

$$I_p = \frac{\lambda}{|P|} + \sum_{q:(q,p)\in L} \frac{(1-\lambda)I_q}{outlinks(q)} + \sum_{q:outlinks(q)=0} \frac{(1-\lambda)I_q}{|P|}$$

where $outlinks(q)$ is the number of links that are outgoing from q. Thus, the sum of the entries is:

$$\begin{aligned}
\sum_{p \in P} I_p &= \sum_{p \in P} \left[\frac{\lambda}{|P|} + \sum_{q:(q,p) \in L} \frac{(1-\lambda) I_q}{outlinks(q)} + \sum_{q:outlinks(q)=0} \frac{(1-\lambda) I_q}{|P|} \right] \\
&= \sum_{p \in P} \frac{\lambda}{|P|} + \sum_{p \in P} \sum_{q:(q,p) \in L} \frac{(1-\lambda) I_q}{outlinks(q)} + \sum_{p \in P} \sum_{q:outlinks(q)=0} \frac{(1-\lambda) I_q}{|P|} \\
&= \lambda + (1-\lambda) \sum_{p \in P} \left(\sum_{q:(q,p) \in L} \frac{I_q}{outlinks(q)} + \sum_{q:outlinks(q)=0} \frac{I_q}{|P|} \right) \\
&= \lambda + (1-\lambda) \\
&= 1
\end{aligned}$$

We can prove that $\sum_{p \in P} \left(\sum_{q:(q,p) \in L} \frac{I_q}{outlinks(q)} + \sum_{q:outlinks(q)=0} \frac{I_q}{|P|} \right) = 1$ as follows. Every page with outlinks distributes all of its PageRank to its neighbors. Furthermore, since the sum of the PageRanks (before entering the loop) is equal to 1, it must be that $\sum_{p \in P} \sum_{q:(q,p) \in L} \frac{I_q}{outlinks(q)} = 1 - \sum_{q:outlinks(q)=0} I_q$. It is easy to see that $\sum_{p \in P} \sum_{q:outlinks(q)=0} \frac{I_q}{|P|} = \sum_{q:outlinks(q)=0} I_q$. Thus, the sum of the two quantities equals 1, completing the proof.

Exercise 5.2. Our model of ranking contains a ranking function $R(Q, D)$, which compares each document with the query and computes a score. Those scores are then used to determine the final ranked list.

An alternate ranking model might contain a different kind of ranking function, $f(A, B, Q)$, where A and B are two different documents in the collection and Q is the query. When A should be ranked higher than B, $f(A, B, Q)$ evaluates to 1. When A should be ranked below B, $f(A, B, Q)$ evaluates to –1.

If you have a ranking function $R(Q, D)$, show how you can use it in a system that requires one of the form $f(A, B, Q)$. Why can you not go the other way (use $f(A, B, Q)$ in a system that requires $R(Q, D)$)?

Given a ranking function $R(Q, D)$, one can construct a pairwise ranking function $f(A, B, Q)$ as follows:

$$f(A, B, Q) = \begin{cases} 1 & R(Q, A) > R(Q, B) \\ -1 & R(Q, A) < R(Q, B) \\ 0 & otherwise \end{cases}$$

The ranking function $f(A, B, Q)$ imposes an ordering on documents with respect to Q. Therefore, it may be possible to assign a *relative* score to the documents scored by $f(A, B, Q)$, but it is not possible to assign an *absolute* score $R(Q, D)$, unless every possible pair of documents was first scored by $f(A, B, Q)$, which is infeasible in practice.

Exercise 5.3. Suppose you build a search engine that uses one hundred computers with a million documents stored on each one, so that you can search a collection of 100 million documents. Would you prefer a ranking function like $R(Q, D)$ or one like $f(A, B, Q)$ (from the previous problem). Why?

For both $R(Q, D)$ and $f(A, B, Q)$, let us determine, in the worst case, the total number of ranking function evaluations necessary to find the K most relevant documents with respect to Q. We first consider the number of evaluations necessary to rank the documents on each individual machine. For $R(Q, D)$, every document needs to be scored, thus resulting in $O(N)$ ($N = 10^6$) ranking function evaluations. Of course, the N documents would need to be sorted, but the time necessary, in most situations, is considerably smaller than the time it takes to compute $R(Q, D)$ for each document. The naive application of $f(A, B, Q)$, where we apply the function to every possible pair of documents requires $O(N^2)$ evaluations of $f(A, B, Q)$. However, if we treat $f(A, B, Q)$ as a comparator, we can use it directly for sorting/ranking, requiring only approximately $O(N \log N)$ evaluations of $f(A, B, Q)$. When it comes to merging the results from each machine, no additional evaluations need to be done when using $R(Q, D)$. However, for $f(A, B, Q)$, we need to compare the top K documents from each machine, resulting in $100K \log 100K$ more evaluations of $f(A, B, Q)$. Therefore, $R(Q, D)$ should be preferred, as it requires fewer ranking function evaluations per machine and does not incur any additional cost when merging results across machines.

Exercise 5.4. Suppose your search engine has just retrieved the top 50 documents from your collection based on scores from a ranking function $R(Q, D)$. Your user interface can show only 10 results, but you can pick any of the top 50 documents to show. Why might you choose to show the user something other than the top 10 documents from the retrieved document set?

There are many reasons why you may want to show the user something other than the 10 documents with the highest score with respect to $R(Q, D)$. For example, you may want to deduplicate the results. Often there are duplicate or near-duplicate documents in a collection and it is typically best to only show one version of the document in the result list. You may also want to show a diverse result set that covers multiple aspects of the query. For example, given the query 'java', it may be best to show results for the programming language, coffee, and the island nation. Other possible reasons include the need to personalize the result set (e.g., make sure results satisfy user preferences) and ensuring fresh results (e.g., only show the freshest, most recent results).

Exercise 5.5. Documents can easily contain thousands of non-zero features. Why is it important that queries have only a few non-zero features?

Most ranking functions, such as the one described in Section 5.2, try to measure the similarity between queries and documents. If a query contains only a few non-zero features, then scoring using inverted indexes is very efficient, because only a small number of features must be considered during scoring. However, if the query contains many thousands of non-zero features, then many feature inverted lists must be opened and, if disjunctive processing is used, many more documents may need to be scored.

Exercise 5.9. In section 5.4.1, we created an unambiguous compression scheme for 2-bit binary numbers. Find a sequence of numbers that takes up more space when it is "compressed" using our scheme than when it is "uncompressed."