What's the Score? Matrices, Documents, and Queries

Dianne P. O'Leary[1]

You have just been given an assignment. As you work through it, you find several terms you don't know, and several assertions you don't understand. What do you do?

Your first instinct is probably to plug one or more of the terms into your favorite search engine and see if one of its suggested webpages answers your questions. The ideal search engine has several properties:

- Retrieval is fast.
- Most of the documents listed at the top are actually relevant to your query. (This is called good **precision**.)
- Few documents at the bottom of the list (which may be millions of documents long) are more relevant than ones listed before them. (This is called good **recall**.)

How do search engines select the documents (webpages) believed to be relevant to your query? They do this using several types of matrix computations. For example, the fundamental tool behind Google is the PageRank algorithm, used to identify authoritative documents on the web. This algorithm is based on finding the eigenvector corresponding to the largest eigenvalue of a matrix whose size equals the number of indexed webpages (several billion!). Most search engines also use a variant of latent semantic analysis (LSA) to choose relevant documents, and in this case study, we'll experiment with this algorithm and alternatives.

[1] This case study is a supplement to *Scientific Computing with Case Studies*, Dianne P. O'Leary, SIAM Press, Philadelphia, 2009.

From Documents to Matrices

Old fashioned search engines used string matching to find relevant documents. Some automated library catalogs[2] still use this technology. If you search for 'Mark Twain', the catalog might give you a list of records that literally contain the two words 'Mark' and 'Twain' in the document or (more likely) in its index entry. Using string matching, a document is chosen based on testing each string of 4 characters in its text record to see if it matches 'Mark', and each string of 5 characters to see if it matches 'Twain'. This is very slow, and it would miss all the records about 'Samuel Clemens', who is the same person.

So we need a better way to represent a document, rather than just using its text. Let's take all of the words in a dictionary[3] and number them from 1 to m. Then we can represent a document as a vector of length m whose ith entry indicates the importance of word i in the document. Further, we can represent a document collection by a matrix $\boldsymbol{A}$ whose jth column is the vector corresponding to document j, for $j = 1, \ldots, n$.

Importance is a concept subject to a lot of argument in the literature. For example, is the word 'the' more important in one document than another, or should we leave it out entirely? Should a document vector be normalized to length 1? (If so, what norm should be used?) Should each row of the document matrix also be normalized? The most popular measure of importance is called TF-IDF. In this case study, we'll assume someone else has made the decision about how to measure importance and has presented us with the **term-document matrix** $\boldsymbol{A}$.

Forming Queries and Scoring Documents

The matrix approach gives us a tremendous speed-up over string matching. If a user asks for a term, we can find the term in the dictionary, get the index i of the term, and decide that document j is relevant if the matrix entry a_{ij} is large. Equivalently, we could compute the relevance of each document in the collection by taking the product $\boldsymbol{q}^T \boldsymbol{A}$, where the **query vector** $\boldsymbol{q} = \boldsymbol{e}_i$ is the ith column of the identity matrix of size m. Similarly, if we are interested in term i and term ℓ, then our query vector is $\boldsymbol{q} = \boldsymbol{e}_i + \boldsymbol{e}_\ell$. This approach is called the **vector space method**, and the relevance evaluation is called the **inner-product score**.

In order to use this method, we need to form a query vector, so we need to be able to find the index of a term in a collection of terms. The function `findtermslow.m`, available on the website, does this, but (as you might guess) it is slow, because it searches sequentially through the collection until it finds a match. If the collection is sorted in alphabetical order, we could do better than this. (When you look in the phone book, you don't start on page 1 and look at every name until you find the one you want!) Let's consider doing the search by **bisection**, a method you may have used previously for finding a zero of a function. We start by asserting that if the word is present, its index must be between 1 and p, where p is the number of words in the dictionary. We then test the word whose

[2] e.g., `http://www.press.uchicago.edu/AAUP/bk.searchguide.html`

[3] For us, the 'dictionary' is a collection of words occurring in our documents.

POINTER 1. Sparse matrices
In this case study we work with a sparse matrix, one that has so many zero elements that it is worthwhile to take advantage of them. MATLAB stores a sparse matrix by remembering the row index, column index, and value of each nonzero entry, so the total amount of storage is about $3nz$, where nz is the number of nonzero entries in the matrix. The Cranfield term-document matrix has 83302 nonzero entries, so the storage for it in sparse format is about 250000 numbers, compared to $4612 * 1398 \approx 6.4$ million numbers for storage in full matrix format. More information about sparse matrices is found in Unit VII.
Use `svds` to compute the SVD of the (sparse) Cranfield matrix. (The function `svd` requires a full matrix.)

---

index i is closest to $p/2$. If our word comes after that word alphabetically, then if present its index must be between i and p; otherwise its index must be between 1 and i. In either case, we have reduced the length of the list by a factor of 2, so we repeat until we either find the term or reduce the list to zero length.

CHALLENGE 1. Write a MATLAB function `findterm` to use bisection to find the index of a term in an alphabetized dictionary list. (The function `Strcmp.m`, available at Matlab Central, might be helpful, although it is very slow, so you may want to improve it.) Your function should be specified as `index = findterm(desired_t termlist, istart, ifinish)`, where the parameters have the same meaning they do in `findtermslow`. Test your function on a random selection of terms from the dictionary in `cranterms.mat`. Generate 500 random indices by taking `floor(rand(500,1)*` where `p` is the number of entries in the dictionary. Compare the time that your function takes to the time that `findtermslow` takes. (MATLAB's `tic` and `toc` can be used.)

Your function should be well designed, well documented, reasonably efficient, and correct. See Chapter 4 for guiding principles on software design and documentation, and use `findtermslow.m` as a model.

Now that we can form query vectors, we are ready to score our documents using the inner-product computation. If $\boldsymbol{q}$ is the query vector, then the largest entries in $\boldsymbol{q}^T\boldsymbol{A}$ identify the documents that are relevant.

But how can we measure how good this method is?

- The good documents are those that are retrieved and also relevant to the query.
- The **precision** $P(\ell)$ is the ratio of the number of good documents to the number ℓ of retrieved documents. (It is low if a lot of irrelevant documents are retrieved.)

- The **recall** $R(\ell)$ is the ratio of the number of good documents to the number of relevant documents. (It is low if a lot of relevant documents are missed.)

A plot of $P(\ell)$ and $R(\ell)$ as a function of ℓ tells us how successful our retrieval is.

As an example, suppose that we have 5 documents, and documents 2, 3, and 5 are relevant to the query. Suppose that your scoring produces $\boldsymbol{q}^T\boldsymbol{A} = [5, 3, 1, 2, 10]^T$. Then if asked to retrieve 1 document, you would return the highest scoring document, number 5 (since it has a score of 10). Document 5 is indeed relevant. So $P(1) = 1$ and $R(1) = 1/3$. If asked to retrieve 2 documents, you would return documents 5 and 1, and 1 is not on the relevant list, so $P(2) = 1/2$ and $R(2) = 1/3$.

We'll try this method on a dataset for which the set of documents relevant to each query is known.

CHALLENGE 2. We'll work with the Cranfield collection, 1398 aerospace engineering abstracts.[4] Score the documents for each of the queries in `cranquerydata.mat` and plot the average values of $P(\ell)$ and $R(\ell)$. (The average is taken over all of the queries, so for each value of ℓ, compute the average value of $P(\ell)$ and $R(\ell)$ and then plot these values vs. ℓ.)

From Term Matching to Concept Matching

By using the term-document matrix, we have made good progress on the speed issue in document retrieval. But we haven't solved the Mark Twain vs. Samuel Clemens problem. This terminology problem is ubiquitous: documents can be in a variety of languages (English, Spanish, etc.), or contain different jargon ('myocardial infarct' vs. 'heart attack').

We could have a person make a list of all synonyms and word relations for words in the dictionary, but this is expensive and time-consuming. Instead we will try to let the documents define their own relations. For example, if there are several documents that contain the four words 'Mark', 'Twain', 'Samuel', and 'Clemens', then we might conclude that these four words have a relationship. How can we use matrix computations to reveal this?

Consider this tiny example of a term-document matrix:

[4] The data is available at `http://www.cs.utk.edu/~lsi/` under "Corpora". The matrix and term list are found toward the bottom of the page, and and software to let MATLAB read a datafile in Harwell-Boeing sparse matrix format is available at `http://people.scs.fsu.edu/~burkardt/m_src/hb_io/hb_exact_read.m`. It is much easier, though, to download the data from the book's website.

Term	D1	D2	D3	D4	D5
mark	20	31	0	5	2
twain	53	65	0	30	1
samuel	5	4	6	10	0
clemens	10	20	40	43	0
europe	30	10	25	52	70

We can see that documents D1 and D2 indicate that 'Mark Twain' and 'Samuel Clemens' are related in some way, so we might conclude that D3 is also about Mark Twain. In particular, D1, D2, D3, and D4 are relevant for the query 'Mark Twain Europe' and for the query 'Samuel Clemens Europe'. If we just use the vector space method, the inner-product scores for the 'Mark Twain Europe' query are 103, 106, 25, 87, 73, so it seems that D5 is more relevant than D3.

One approach to fix this problem is to replace $\boldsymbol{A}$ by an optimal approximation $\boldsymbol{A}_k$ of the same size as $\boldsymbol{A}$ but of rank $k << \min(m, n)$. We know from Chapter 5 that the optimal approximation is formed by taking $\boldsymbol{A}_k$ equal to the the truncated SVD. In fact, $\boldsymbol{A} = \boldsymbol{A}_k + \boldsymbol{E}$ where $\|\boldsymbol{E}\| = \sigma_{k+1}$, the largest singular value that we threw away. So forming $\boldsymbol{q}^T \boldsymbol{A}_k$ just gives us an error of $\boldsymbol{q}^T \boldsymbol{E}$, which is small. The effect of the approximation is to add **noise** to the matrix to make the columns look more alike, so that there will be no more than k linearly independent columns in any selection of columns. Intuitively, the smallest amount of noise is added when columns that are related to each other are made even more dependent, so truncating the matrix reveals relationships that are latent in the documents. This approach is called **latent semantic analysis (LSA)**. The size of k is determined based on two criteria:

- We don't want k too small, because in that case document vectors will look too much alike and have too much noise.
- We don't want k too large, because m and n are generally large, so storage is at a premium. If we store $\boldsymbol{U}_k$, $\boldsymbol{V}_k$, and $\boldsymbol{\Sigma}_k$, we need $(m + n)k + k$ storage locations, so the smaller we can keep k, the better off we are. In fact, $\boldsymbol{A}$ is probably sparse, but the factors are generally dense, so a small k is essential. A small k also makes document scoring cheaper.

  Let's see how well this works.

CHALLENGE 3. Replace $\boldsymbol{A}$ by $\boldsymbol{A}_k$ and recompute the average precision and recall curves. Be sure that you implement the method so that the time to compute $\boldsymbol{q}^T \boldsymbol{A}_k$ is proportional to $(m + n)k + k$, not mn. Use values $k = 100$, $k = 500$, and others of your choosing. Compare your results to those using the vector space method. Write a clear paragraph of advice about how to choose k.

There are other (cheaper) ways to compute a low-rank approximations to a matrix. Can they also be used for document retrieval?

POINTER 2. Quirks in Data and Software
For query 222 in this dataset, some of the documents labeled relevant are not within the valid range of document indices. One of the "features" of working with real data is that it is never perfect, and programs need to allow for that.
Software also has its quirks. If you compute `[Q,R,P] = qr(A)` for a sparse matrix `A`, MATLAB will report "too many output arguments". MATLAB's `qr` requires a dense matrix if a pivoted QR factorization is computed, so if you need to do it, use `[Q,R,P] = qr(full(A))`.

CHALLENGE 4. Choose another algorithm for computing a low-rank approximation to $\boldsymbol{A}$ and repeat the experiment of the previous challenge using this algorithm. Compare the results and write a clear paragraph of advice about how to choose k and what method you recommend.

Even though we understand low-rank matrix approximation, the reason that LSA works (and sometimes doesn't work) is a bit mysterious. Working with our tiny example might yield insight.

CHALLENGE 5. Consider the Mark Twain term-document matrix given above. It would be ideal to have a low-rank approximation that for the sample query scores D1, D2, D3, and D4 high but not D5. Discuss what properties of the low-rank SVD prevent this, and illustrate your discussion with some examples of other small term-document matrices for which the partitioning is more clear.

Related Methods and Open Questions

The LSA approach has truly revolutionized information retrieval, but there is much unfinished work. SVD and other low-rank approximation methods from Chapter 5 are powerful tools, but they have some disadvantages for this application:

- Matrix entries in $\boldsymbol{A}_k$ can be negative, and it is hard to interpret a negative importance.
- Factors that are as sparse as $\boldsymbol{A}$ would be more economical.

POINTER 3. Further Reading.
To learn more about the uses of matrices and matrix factorizations in document retrieval and data mining, see [2]. The semi-discrete decomposition is considered in [3]. An example of a complete document retrieval system based on matrix computations and optimization is given in [1].

[1] Daniel M. Dunlavy, Dianne P. O'Leary, John M. Conroy, and Judith D. Schlesinger, "QCS: A System for Querying, Clustering, and Summarizing Documents," Information Processing and Management 43:6 (2007), 1588–1605. DOI:10.1016/j.ipm.2007.01.003.

[2] Lars Eldén, *Matrix Methods in Data Mining and Pattern Recognition*, SIAM Press, Philadelphia, PA, 2007.

[3] Tamara G. Kolda and Dianne P. O'Leary, "Computation and Uses of the Semidiscrete Matrix Decomposition," ACM Transactions on Mathematical Software, 26 (2000) 415-435. http://portal.acm.org/citation.cfm?DOId=358407.358424.

- Computing the SVD (or even the rr-QR factors) of a matrix with billions of rows is expensive!

So alternative approaches are also used, such as the semi-discrete decomposition, non-negative matrix approximations and approximations consisting of a subset of the rows or columns of $\boldsymbol{A}$.

In our query vectors, we gave each term a weight of 1. Better results would be obtained by setting the importance of each term in a better way, but there is no consensus on how this should be done.

An alternative to the inner product score is the **cosine score**. We compute the cosine score for a document vector $\boldsymbol{a}_j$ as $\boldsymbol{q}^T\boldsymbol{a}_j/(\|\boldsymbol{q}\|\,\|\boldsymbol{a}_j\|)$. This is the cosine of the angle between the vectors $\boldsymbol{a}_j$ and $\boldsymbol{q}$, and it avoids giving long documents an unreasonably large score.

Conclusions

By now you have seen some of the issues that arise in computational science collaborations.

- Each field has its own jargon (and cultural issues) that must be mastered.
- In contrast to textbook problems that give you an equation to solve, a great deal of effort is devoted to figuring out what the problem actually is and deciding what equations and techniques are appropriate.
- The measure of "success" is also different for each field (e.g., precision and recall).

- Finding test data and learning to work with it takes a lot of time. This is especially true if the data is obtained from the internet sites listed here, rather than from our own website. It took several hours to make the `.mat` files for this project. First, it took a while to find the appropriate software to convert the matrix from Harwell-Boeing format to MATLAB format. (As usual, advice from a colleague was helpful.) Then, there were two inconsistencies in the file format. I fixed one by changing the datafile and one by removing an `if` statement in a conversion routine. Data issues always cause big headaches in collaborative projects.

- It is impossible to be an expert on every aspect of the problem, and you must rely on other members of the team to judge correctness and importance.

Once you understand these facts, you are well on your way to understanding the job of a computational scientist! There is always something new to learn, and it is never boring.