Evaluation of Document Ranking Methods Involving Matrix Factorization

David (Jung Won) Yang

jwy273

Abstract

In modern day document databases, there can be millions if not billions of documents. In order to make use of the abundance of information we need effective methods of document ranking or information retrieval so that we can easily access relevant information. Primitive methods of term matching was used in the past, however, this method is computationally costly for large databases, and does not perform very well. Later, a new approach relying to information retrieval relying on linear algebra was developed. This method is called the vector space method, and it represents documents as vectors in a vector space. In order to improve the vector space method latent semantic analysis and dimensional reduction was developed. The idea behind these methods is to bring out the latent structure between the documents. Ideally documents with related semantic meaning will have dependent latent structure. Test each of these methods on the Cranfield data set, and evaluate their performances.

I. Introduction

Following the advancement of digital technology, there has been a growth of digital databases holding text data such as documents. The growth of the internet has lead to data bases that hold millions if not billions of documents. To make use of the massive amount of data and information we require good techniques and methods of information retrieval. The initial primitive method of text retrieval were methods that used string matching. In string matching, we go through each document looking to see if a particular phrase or term exists in the document. The concept of string matching follows the same idea as the Boolean framework mention in A.B. Manwar's paper[1]. In the Boolean frame work, a query is compared with the documents by matching the query with the terms or phrases in the documents. The matching is evaluated through the Boolean operators AND, OR, NOT, and the documents are predicted to be either relevant or non-relevant. String matching or the Boolean framework has very unsatisfactory performance in terms of efficiency, and accuracy. When string matching, you must compare your term or phrase with every string in the document that is of the same length which is very expensive. In addition, the method is unable to do partial matching as the documents are either predicted to be relevant or irrelevant. This often leads to very few documents to be selected, or for the selection to favor large documents. A new method called the Vector Space method was discovered by G. Salton in the late 1960s [1], and this method brought a whole different approach to information retrieval, and ranking documents. The Vector Space method is a big improvement to the string matching method in regards to both efficiency and accuracy. In the vector space method, the documents are stored in a matrix, called the term document matrix, where each column corresponds to a document, and the rows correspond to the terms in a determined list of terms or dictionary. We now work with matrices instead of text data, and we utilize linear algebra instead of simple text matching. However, there are certain issues that must still be addressed. The two fundamental issues for indexing and ranking of documents is polysemy and synonymy. Polysemy is when a single term has multiple meanings, and synonymy is when a several terms have the same meaning. In addition to these issues, relevant documents may simply not contain the same words as those entered in the query, but may also be similar in terms of content. In order

to address this issues, a method called latent semantic analysis is used. As the name suggests, the method brings out the latent structure of the documents. This is done by factoring the matrix in order to create a lower rank approximation of the term document matrix. We then use the factored matrix as the term document matrix. Rank reduction is one option. another about is to us the truncated SVD to create a lower dimensional representation of the term document matrix and query vector. In this project, we implement the original vector space method, latent semantic analysis, and dimensional reduction. We run the methods on a data set to evaluate and compare the methods.

II. Vector Space Method

In the vector space method we use linear algebra in order to predict the relevance between the documents in a database, and a given query. The documents are stored in a matrix called the term document matrix. In the term document matrix, the each column corresponds to a document and each row corresponds to a row. An entry in a column, is the importance of the corresponding word in the document. So entry i,j of the term document matrix would be the importance of the ith term in the jth document. We assume that the documents and the terms have been given an ordering and an index. There are many ways to determine the importance of a word in a document. One simple measure of importance in a document is to count the number of times a certain term appears. If m is the number of terms in our dictionary or term list, and n is the number of documents, the size of the term document matrix should be mxn.

We also turn the query into an object that we can utilize for mathematical operation. The query comes in the form of a string that contains a list of terms. For each term, we get the corresponding dictionary index of the term (we use dictionary as simply a term list). Let m be the number of terms in our dictionary. We turn the query into a m dimensional vector that is also zeros except for the entries that correspond to the dictionary indices of the terms from the query, those entries have a value of one. For example, if the query contains term x, and this term corresponded to the jth term in the term list. Then the jth entry of the query vector should equal 1 and the rest of the entries should equal 0.

So now the question is , how to we compute the relevance of the documents? We do so by computing a score for each document which quantifies the relevance of each document to the given query. We get the score of a document by computing the dot product between the query vector and the document.
Let q be the query vector, and let d_j be the jth column of the term document matrix.

$$score = q^T d_j$$

Let A be the term document matrix. To get the scores of every document we simply replace d_j with A

$$score = q^T A$$

The score is not a binary value, but a continuous value that quantifies the degree of similarity between the query and a document. If we want to normalize the score by the size of the document so that larger documents do not have an advantage, we can compute the cosine of the angle between the query vector and the document column vector instead of the dot product.

$$score = \frac{q^T d_j}{|q||d_j|}$$

After we have the scores of the documents, we can rank the documents by their score.

III. Latent Semantic Analysis

In Latent Semantic Analysis we use matrix factorization to find a lower dimensional representation of the term document matrix. Due to the way that the document vectors are constructed, the different columns are

generally generally have very little dependence for each other. By constructing a lower rank approximation, we seek to make related columns more similar or dependent on one another.

Let A be a matrix of rank n and let A_k be a matrix of rank $k << n$. Consider the approximation error

$$E = ||A - A_k||_F$$

Intuitively, the smallest error will be achieved if we reduce the rank of A by making similar columns more dependent on each other [2]. This method follows a certain ideology of semantics in natural language process. In this ideology, we believe that words that are related in meaning will appear among similar distributions of words.

We reduced the rank of the matrices using two different matrix factorizations. The first matrix factorization used is singular value decomposition.

$$E = ||A - A_k||_F = min_{rank(X)\leq k}||A - X||_F = \sqrt{\sigma_k^2 + 1 + ... + \sigma_n^2} \quad [3]$$

Where n is the rank of A. From the theorem by Eckart and Young, it is proven that the error E is minimized when A_k equals the truncated SVD. Let the SVD of A equal USV^T and let the truncated SVD equal $U_k S_k V_k^T$ where U_k contains the first k columns of U, S is a diagonal matrix that contains the first k singular values, and V_k contains the first k columns of V. We assume that the singular values are in descending order.

The truncated SVD may be the best approximation in terms of error, however there are two key features of the term document matrix that the truncated SVD fails to preserve. The first is non-negativity of the entries and the second is sparsity. Each entry in the matix signifies the importance of a term in a document, and thus it makes sense to have strictly non-negative values for the entries. Unlike the term document matrix, A, the low rank approximation given by the truncated SVD has negative entries, and the matrix is very dense. The negative values on the entries are an issue because we do not have an interpretation for negative values in our model. The fact that the truncated SVD is represented as a matrix factorization, where each of the factors, U_K, S_K, and V_k, are significantly smaller than A, makes the computation of the score much more efficient than multiplying by dense matrix that is of the same size as A. However, A is a very sparse matrix, which makes matrix multiplication much more efficient. For this reason, when the value of the dimension, k, gets relatively large, the computation of the score using the truncated SVD becomes more expensive than the computation of the score using the sparse matrix A. In hopes to address the issues of the truncated SVD, another factorization was considered, namely non-negative matrix factorization. Non-negative matrix factorization gives us a low rank approximation where the entries are all non-negative, and it is claimed that when you factor a sparse matrix, the factors tend to be sparse as well.[5] An problem with methods that compute non-negative matrix factorization is that the methods no not ensure optimal error like the truncated SVD, and only factorization of rank that is less than k can be guaranteed. It is often the case that the resulting approximation has a rank less than k.

IV. Dimensional Reduction

In the LSA that we described, we use the truncated SVD to get a low rank approximation of the term document matrix, A_k, and then use the matrix A_k instead of the term document matrix, A, to compute the scores for the documents. Not only can we use the truncated SVD to compute a low rank approximation, but we can also use it to constructed a lower dimension representation of the term document matrix and the query vectors.[4] If we recall, the truncated SVD has the form $U_k S_k V_k^T$. We consider the columns in V_k to be lower dimensional approximations to the columns in A. Note that V_k^T is a k by n matrix. So the columns of V_k^T are k dimensional, where as the column vectors in A are m dimension (m being the number of terms in our term list or dictionary). We use V_k^T as a low dimensional representation of the term document matrix. In order to compute the score we must also project the query vector to the same space as V_k^T. Let q be the

query vector, the projection of the query vector onto the space of V_k^T is given by

$$q' = U_k^T S_k^{-1};$$

After we compute the projected query vector, we compute the scores for the documents with the familiar equation.

$$score = {q'}^T V_k^T$$

What dimensional reduction seeks to achieve is similar to what LSA seeks to achieve, and that is to bring out the latent structure in the documents. Like LSA, we expect the latent structure to help remove the issues of polysemy and synonymy. The idea is that two terms with the same semantic meaning will generally be among the same distribution of terms. So it is expected that terms or documents with words that have related semantic value will have related representations in the reduced space.

V. Experiment

A) Data set

The data set that we use is the Cranfield collection. This data set contains 1398 aerospace engineering abstracts, and a dictionary that contains 4612 terms. The data set also contains a list of queries, and a list of documents that are relevant to the query. This data set was obtained in the website for Dianne P. O'leary's book, Scientific Computing with Case Studies. [2]

B) Implementation

All implementation was done using Matlab.

1) Preprocessing the Data: The data was obtained in .mat files. The term document matrix, and the term list was extracted from the structures in order to make them easier to work with.

2) Implementation:

findterm: There was another function given in website for Dianne P. O'leary's book that also returned the index of a term in a term list. However, the performance of the function was slow as it used linear search. The function implemented uses bisection. As expected it performs significantly better, even for terms near the front of the list.

Strcmp: Matlab's built in string comparison function does not give us the lexicographical ordering of the terms, so another method for string comparison was implemented.

Sequence of Execution
1. Computer the truncated SVD and the non-negative matrix factorization of the term document matrix A.
2. For each query, generate a query vector that contains the indices of the words in the query.
3. For each query compute both the score and the cosine score for each document using the original term document matrix, the two low rank approximations, and LSI.
4. Sort the array documents in descending order of their score
5. Take the top n documents, where n is a selected number.
6. Computer the precision and recall for each query, and then compute the average recall and average precision.

VI. Testing

When evaluating the performance of document ranking or information retrieval, the two important metrics are recall and precision. Recall measures the percentage of relevant documents that were retrieved, and precision measure the number of relevant documents among the documents that were retrieved.

We test the precision and recall of the queries for three different ranks, and for three different values of n where n is the number of documents that we observe after we rank the documents. We observe the n documents with the highest score. We also plot the change in precision and recall for different values of n, and for different values for k. We want to compare the performances of the different methods and matrix factorization, and we also want to see the change in the recall and precision as we change the value of k.

VII. Results

Recall using dot product for the score.

k = 100

n	vs	svd	nnmf	dim red
10	0.183949	0.143100	0.131147	0.159480
20	0.263539	0.214187	0.194094	0.240446
30	0.300653	0.266461	0.234494	0.289022

k = 200

n	vs	svd	nnmf	dim red
10	0.183949	0.164918	0.138363	0.178095
20	0.263539	0.237248	0.199711	0.254105
30	0.300653	0.290308	0.251488	0.295862

k = 300

n	vs	svd	nnmf	dim red
10	0.183949	0.175421	0.143084	0.175152
20	0.263539	0.250996	0.214701	0.231465
30	0.300653	0.304057	0.264771	0.279614

k = 400

n	vs	svd	nnmf	dim red
10	0.183949	0.175875	0.149355	0.174320
20	0.263539	0.257005	0.230173	0.236425
30	0.300653	0.305515	.283147	0.268428

Recall using cos angle for the score.

k = 100

n	vs	svd	nnmf	dim red
10	0.188963	0.142804	0.130573	0.158906
20	0.254663	0.213910	0.193519	0.239872
30	0.303486	0.267247	0.233920	0.288171

k = 200

n	vs	svd	nnmf	dim red
10	0.188963	0.164344	0.137789	0.177521
20	0.254663	0.236970	0.199137	0.252956
30	0.303486	0.289734	0.250914	0.294714

k = 300

n	vs	svd	nnmf	dim red
10	0.188963	0.174847	0.143084	0.174004
20	0.254663	0.250422	0.214127	0.230316
30	0.303486	0.303483	0.264197	0.278466

k = 400

n	vs	svd	nnmf	dim red
10	0.188963	0.175301	0.149355	0.173450
20	0.254663	0.256727	0.229598	0.235277
30	0.303486	0.304941	0.282573	0.267280

Precision using dot product for the score.

k = 100

n	vs	svd	nnmf	dim red
10	0.138222	0.114667	0.103111	0.124889
20	0.102889	0.086444	0.079111	0.095556
30	0.079111	0.072000	0.064296	0.075852

k = 200

n	vs	svd	nnmf	dim red
10	0.138222	0.129333	0.112444	0.133333
20	0.102889	0.094444	0.083111	0.096889
30	0.079111	0.076889	0.068593	0.075259

k = 300

n	vs	svd	nnmf	dim red
10	0.138222	0.135556	0.116000	0.132889
20	0.102889	0.098444	0.088667	0.087111
30	0.079111	0.079852	0.071704	0.070370

k = 400

n	vs	svd	nnmf	dim red
10	0.138222	0.137778	0.120000	0.130667
20	0.102889	0.099333	0.091333	0.088889
30	0.079111	0.080000	0.074667	0.067111

Precision using cos angle for the score.

k = 100

n	vs	svd	nnmf	dim red
10	0.144889	0.114222	0.102222	0.124000
20	0.098222	0.086222	0.078667	0.095111
30	0.079111	0.072148	0.064000	0.075407

k = 200

n	vs	svd	nnmf	dim red
10	0.144889	0.128444	0.111556	0.132444
20	0.098222	0.094222	0.082667	0.096000
30	0.079111	0.076593	0.068296	0.074667

k = 300

n	vs	svd	nnmf	dim red
10	0.144889	0.134667	0.116000	0.131111
20	0.098222	0.098000	0.088222	0.086222
30	0.079111	0.079556	0.071407	0.069778

k = 400

n	vs	svd	nnmf	dim red
10	0.144889	0.136889	0.120000	0.129333
20	0.098222	0.099111	0.090889	0.088000
30	0.079111	0.079704	0.074370	0.066519

Figure at the end of the document

When we look at the change in precision and recall as we increase n, the number of documents we observe, we see that the recall increases, and the precision decreases. Also the difference in recall between n = 10 and n = 20 is larger than n = 20 and n = 30. This means that relevant documents are more likely to be towards the front of the ranking, and thus the concentration of relevant documents decrease as we move towards the end of the ranking.

If you look at the figure at the end of the document, there are plots that show the change in recall and precision as we increase k which is the rank in the case of LSA, and the dimension in the case of dimension reduction. For this plot, we have fixed the value of n (the number of documents retrieved) to 20. When comparing the performance for the document ranking using the dot product score and the cos angle, we see that the results are rather similar. Interestingly, the vector space method always performed better than the other methods when using the dot product to compute score. However, when using cosine angle to compute score, LSA using the truncated SVD performed better than vector space method for higher values of k.

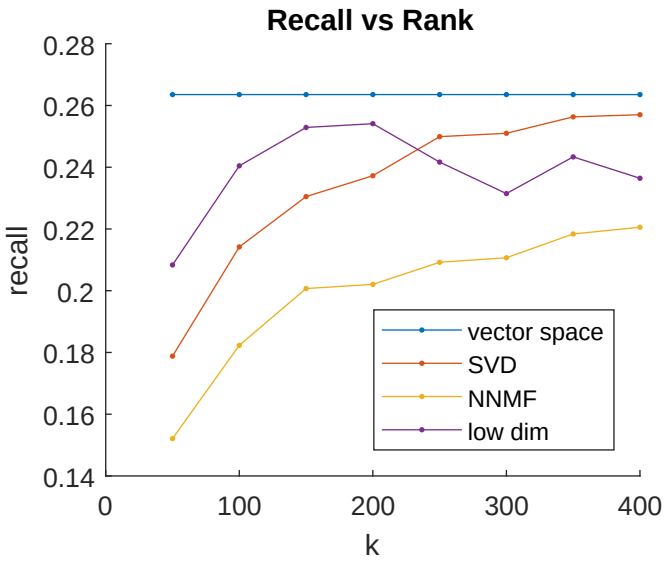
When increasing the value of k, the method using dimensional reduction followed a very different trend than the LSA methods. Dimension reduction performed much better than the LSA methods for the low values of k. However, unlike the LSA methods which increased uniformly as we increased k (although the precision slightly decreased at the end), dimensional reduction performed best at a value of k between 150 and 200. There is no definitive optimal value for k, and the value is usually selected empirically. Generally values of k in the low hundreds are known to be good values, and our results support this claim. Looking at the graph, we see that the recall and precision recall peak around 350, and increasing k leads to at most minimal improvement and is not worth the extra computation cost.

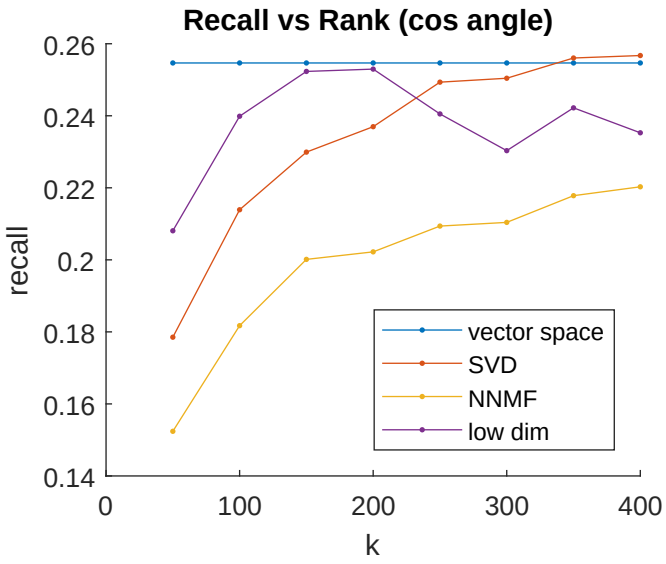
Non-negative matrix factorization has attractive properties, which are the non-negativity of the entries, and the possible sparsity. For our data set, LSA using non-negative matrix factorization always performed worse than the truncated SVD. Both matrix factorization improved in recall and precision as k was increase, however, if you observe the plot, you see that NNMF is always below the truncated SVD, and the difference is quite significant. The non-negative factorization of the term document matrix had many zero entries, however, it was not sparse enough to take advantage of.

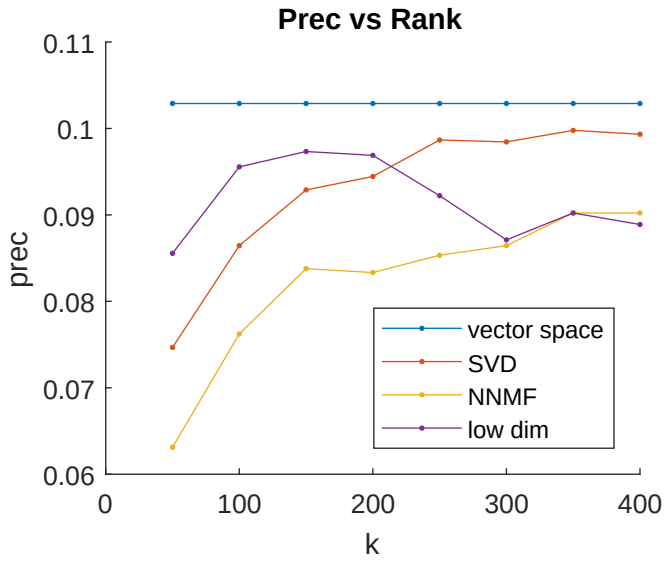
When only 10 documents are retrieved we see that there is a bigger difference in precision and recall between the vector space method and the other methods, especially for lower values of k. This means that the vector space model was able to rank more relevant documents towards the front of the ranking. We see that as the number of documents retrieved increase, the difference between the vector space method and the other methods decrease. We also tested the precision and recall for very numbers of documents retrieved and eventually the recall and precision of the truncated SVD pass the vector space method.

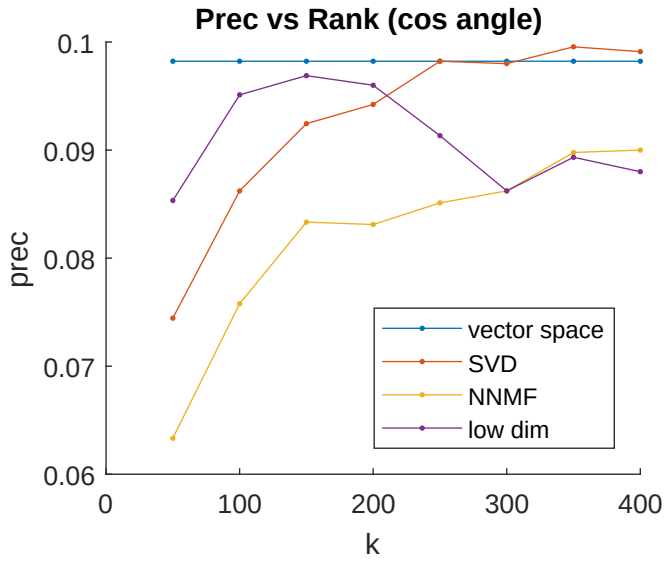
VIII. Discussion

When testing our implementation of document ranking on the Cransfield collection, we see that the vector space method had the best performance. LSA using the truncated SVD was able to achieve higher precision and recall than the vector space method for higher values of k and for higher numbers of documents retrieved. However, the improvements were too small to justify the extra computation cost for the SVD. Not only is computing the truncated SVD expensive, but computing score is expensive as well. The computations are especially expensive for the higher values of k. When k = 300, we found that the computation of the score for LSA, using SVD or NNMF, took more than 5 times longer than the vector space method. Computation complexity of the SVD is an issue for extremely large data sets of millions of documents. Thus other approximation techniques of the SVD or other rank reduction techniques have to be used in order to use LSA. Latent semantic analysis, and dimension reduction is certainly known to be very useful for certain tasks and problems in information retrieval, and for certain data sets. However, we have seen that it is not always the case that the techniques are beneficial. These techniques rest on the assumption that latent structure is reveal by rank or dimension reduction, and thus the lower rank/ dimension representation is the true or better representation of the term document matrix. However, these assumptions are clearly not always true or they do not always show in the results. Rank reduction and the dimensional reduction can bring out latent structure, however, it can also lead to information loss and general inaccuracies in the representation of the term document matrix (which is why we need an adequately high value of k). This loss in information may be the reason that the non-negative matrix factorization did considerable worse than the truncated SVD, as it does not guarantee an optimal low rank approximation like the truncated SVD.


Recall vs Rank
recall
k
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0
100
200
300
400
vector space
SVD
NNMF
low dim


Recall vs Rank (cos angle)
recall
k
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0
100
200
300
400
vector space
SVD
NNMF
low dim


Prec vs Rank
prec
k
0.11
0.1
0.09
0.08
0.07
0.06
0
100
200
300
400
vector space
SVD
NNMF
low dim


Prec vs Rank (cos angle)
prec
k
0.1
0.09
0.08
0.07
0.06
0
100
200
300
400
vector space
SVD
NNMF
low dim

Source Code

Compute the recall:

```
function recall = getRecall(docs,relevant_docs)
%This function returns the recall rate for the document ranking.

%Input: docs is an array that contains the indices for the top ranked
%documents of the query.
%relavent_docs is an array that contains the indices of the docs that are
%relavent to the query.

%Output: recall is the calculated recall rate of the document ranking.

    %Among the documents that were retrieved, we count the number of
    %documents that are relevant.
    n = 0;

    %Count the number of relevant documents in the array, docs
    for i = 1:length(docs)
        if ismember(docs(i),relevant_docs)
            n=n+1;
        end
    end
    %Return the recall
    recall =  n/length(relevant_docs);
end
```

Compute the precision:

```
function prec = getPrecision(docs,relevant_docs)
%This function returns the precision of the document ranking.

%Input: docs is an array that contains the indices for the top ranked
%documents of the query.
%relavent_docs is an array that contains the indices of the docs that are
%relavent to the query.

%Output: prec is the calculated precision for the document ranking.

    %Among the documents that were retrieved, we count the number of
    %documents that are relevant.
    n = 0;
    %Count the number of relevant documents in the array, docs
    for i = 1:length(docs)
        if ismember(docs(i),relevant_docs)
            n=n+1;
        end
    end
    %Return the precision
    prec =  n/length(docs);
end
```

Find the index of a term in a term list:

```
function index = findterm(desired_term, termlist, istart, ifinish)
%This function searches for string among a collection of strings using
%bisection. It assumes that the collection of strings are ordered.

% On input:
%
%   desired_term = the string for which we search.
%   termlist is a structure, with the collection of terms
%            stored as termlist(k).term, k=1:length(termlist).
%   istart = index in termlist from which the search should begin.
%            The default value is 1.
%   ifinish = index in termlist at which the search should end.
%            The default value is length(termlist).
% On output:
%
%   index   = the index of the desired term in the termlist.
%   index   = 0 if the term is not found among terms istart through
%                 ifinish.

% If ifinish is not specified, we search to the end of the collection.

if (nargin < 4)
   ifinish = length(termlist);
end

% If istart is not specified, we search from the beginning of the collection.

if (nargin < 3)
   istart = 1;
end

desired_term = lower(desired_term);

% Search the collection using bisection.
index = floor((ifinish+istart)/2);

found = false;

while ( ifinish >= istart)
    if Strcmp(desired_term, termlist(index)) == 0
        return;
    elseif Strcmp(desired_term, termlist(index)) > 0
        istart = index+1;
    else
        ifinish = index -1;
    end

    index = floor( (ifinish+istart)/2 );
end
```

```
50
51 %If the term was not found return an index of -1
52 if ~found; index = -1; end
53
54 end
```

Compare two strings:

```
1  function cmp = Strcmp(str1,str2)
2  %cmp = 1 if str1 is lexicographically greater than str2
3  %cmp = -1 if str 2 is lexicographically greater than str1
4  %cmp = 0 if they are equal
5
6      len = min(length(str1),length(str2));
7      i = 1;
8      while i <= len
9          if str1(i) > str2(i)
10             cmp = 1; return
11         elseif str1(i) < str2(i)
12             cmp = -1; return
13         end
14         i = i+1;
15     end
16
17     if strlength(str1) == strlength(str2)
18         cmp = 0;
19     elseif strlength(str1) > strlength(str2)
20         cmp = 1;
21     else
22         cmp = -1;
23     end
24 end
```

Compute the norm for each document:

```
1  function v = get_doc_norms(A)
2  %This function calculates the norm for document vectors.
3
4  %Input: A  is a term document matrix, where the columns represent the
       documents.
5
6  %Output: v is a vector that contains the norm for each column.
7
8      ndocs = size(A,2);
9      v = ones(ndocs,1);
10
11     for i = 1:ndocs
12         v(i) = norm(A(:,i));
13     end
14
15 end
```

Compute the score for each document given a query:

```
function score = score_docs(query, A, method)
    %Input: query is a query vector, A is a term document matrix, method
    %specifies the method used to compute the score.

    %Output: score is a vector that contains the score for each document.

    if method == "vs"; score = score_docs_vs(query , A);
    elseif method == "svd"; score = score_docs_svd(query , A{1},A{2},A{3});
    elseif method == "nnmf"; score = score_docs_nnmf(query,A{1},A{2});
    elseif method == "lsi"; score = score_docs_lsi(query,A{1},A{2},A{3});
    end

end

function score = score_docs_vs(query, A)
    nterms = size(A,1);
    query_vec = zeros(1,nterms);
    query_vec(query) = 1;
    %disp(timeit( @() query_vec*A));
    score = query_vec*A;
    score = score';
end

function score = score_docs_svd(query, U, S, V)
    nterms = size(U,1);
    query_vec = zeros(1,nterms);
    query_vec(query) = 1;
    %disp(timeit( @() query_vec*U*S*V'));
    score = query_vec*U*S*V';
    score = score';
end

function score = score_docs_nnmf(query, W, H)
    nterms = size(W,1);
    query_vec = zeros(1,nterms);
    query_vec(query) = 1;
    %disp(timeit( @() query_vec*W*H));
    score = query_vec*W*H;
    score = score';
end

function score = score_docs_lsi(query, U, S, V)
    nterms = size(U,1);
    query_vec = zeros(1,nterms);
    query_vec(query) = 1;
    query_vec = (inv(S)*(U')*query_vec')';
    %disp(timeit( @() query_vec*U*S*V'));
    score = query_vec*V';
    score = score';
```

```
end
```

Compute the cos score:

```
function score = cos_score(query, A, doc_norms, method)
    %Input: query is a query vector, A is a term document matrix, method
        specifies the method
    %used to compute the score, doc_norms is a vector that contains the norm
        for each document.

    %get the dot product scores for the documents.
    score = score_docs(query, A, method);
    query_norm = norm(query);
    ndocs = size(A,2);

    %divide each dot product score  of each document by norm of the query
        vector times the
    %norm of the document.
    for i = 1:ndocs
        score(i) = score(i)/(query_norm*doc_norms(i));
    end
end
```

Convert list of string to a query vector:

```
function query_index = string_to_index(query,termlist)
    %Input: query is a list of strings which are the terms in the query,
    %termlist is a list of terms that has a certain ordering.

    %output: query_index is a list which contains the indices of the
    %terms in the query. The indices correspond to the indices of the terms
    %in the term list.

    n = 1;
    query_index = [];
    for i = 1:length(query)
        %get the index of the term
        index = findterm(query(i),termlist);
        %if the term is not in the term list, ignore
        if (index ~= -1)
            query_index(n) = index;
            n = n+1;
        end
    end
end
```

Tests:

```
function test_LSA_VectorSpace()
```

```
2       term_doc_matrix = load('cranmatrix.mat');
3       querydata_ = load('cranquerydata.mat');
4       termlist_ = load('cranterms.mat');
5
6       A1 = term_doc_matrix.A;
7
8
9
10      [termlist,querydata] = modify_termlist_querydata(termlist_,querydata_);
11      nqueries = querydata_.nqueries;
12
13      %Initialize the arrays that will store the precision for each query.
14      prec = zeros(nqueries,1);
15      prec_svd = zeros(nqueries,1);
16      prec_nnmf = zeros(nqueries,1);
17      prec_lsi = zeros(nqueries,1);
18
19      prec_cos = zeros(nqueries,1);
20      prec_svd_cos = zeros(nqueries,1);
21      prec_nnmf_cos = zeros(nqueries,1);
22      prec_lsi_cos = zeros(nqueries,1);
23
24      %Initialize the arrays that will store the recall for each query.
25      recall = zeros(nqueries,1);
26      recall_svd = zeros(nqueries,1);
27      recall_nnmf = zeros(nqueries,1);
28      recall_lsi = zeros(nqueries,1);
29
30      recall_cos = zeros(nqueries,1);
31      recall_svd_cos = zeros(nqueries,1);
32      recall_nnmf_cos = zeros(nqueries,1);
33      recall_lsi_cos = zeros(nqueries,1);
34
35      avg_recall = zeros(4,3,4);
36      avg_recall_cos = zeros(4,3,4);
37      avg_prec = zeros(4,3,4);
38      avg_prec_cos = zeros(4,3,4);
39
40
41      %Compute the norm/length of each document.
42      doc_norms = get_doc_norms(A1);
43
44      diary('results.txt');
45
46      k = [50,100,150,200,250,300,350,400];
47
48      for j = 1:length(k)
49          [U,S,V] = svds(A1,k(j));
50          A2 = {U,S,V};
51
52          [W,H] = nnmf(A1,k(j));
53          A3 = {W,H};
```

```
54
55          fprintf("-----------------------------------------------\n");
56          fprintf("k = " + string(k(j)) + "\n");
57
58          n = [5,10,15,20,25,30];
59          for m = 1:length(n)
60              for i = 1:nqueries
61                  %Turn the list of strings from the query into a query vector
                        of
62                  %indices.
63                  query = querydata.words{i};
64                  query_index = string_to_index(query,termlist);
65
66                  scores = score_docs(query_index, A1, "vs");
67                  scores_svd = score_docs(query_index, A2, "svd");
68                  scores_nnmf = score_docs(query_index, A3, "nnmf");
69                  scores_lsi = score_docs(query_index, A2, "lsi");
70
71                  cos_scores = cos_score(query_index,A1,doc_norms,"vs");
72                  cos_scores_svd = cos_score(query_index,A2,doc_norms,"svd");
73                  cos_scores_nnmf = cos_score(query_index,A3, doc_norms, "nnmf")
                        ;
74                  cos_scores_lsi = cos_score(query_index,A2,doc_norms,"lsi");
75
76                  [sorted_scores,indices] = sort(scores,'descend');
77                  [sorted_scores_svd,indices_svd] = sort(scores_svd,'descend');
78                  [sorted_scores_nnmf,indices_nnmf] = sort(scores_nnmf,'descend'
                        );
79                  [sorted_scores_lsi,indices_lsi] = sort(scores_lsi,'descend');
80
81                  [sorted_scores_cos,indices_cos] = sort(cos_scores,'descend');
82                  [sorted_scores_svd_cos,indices_svd_cos] = sort(cos_scores_svd,
                        'descend');
83                  [sorted_scores_nnmf_cos,indices_nnmf_cos] = sort(
                        cos_scores_nnmf,'descend');
84                  [sorted_scores_lsi_cos,indices_lsi_cos] = sort(cos_scores_lsi,
                        'descend');
85
86                  prec(i) = getPrecision(indices(1:n(m)),querydata.relevantdocs{
                        i});
87                  prec_svd(i) = getPrecision(indices_svd(1:n(m)),querydata.
                        relevantdocs{i});
88                  prec_nnmf(i) = getPrecision(indices_nnmf(1:n(m)),querydata.
                        relevantdocs{i});
89                  prec_lsi(i) = getPrecision(indices_lsi(1:n(m)),querydata.
                        relevantdocs{i});
90
91                  prec_cos(i) = getPrecision(indices_cos(1:n(m)),querydata.
                        relevantdocs{i});
92                  prec_svd_cos(i) = getPrecision(indices_svd_cos(1:n(m)),
                        querydata.relevantdocs{i});
93                  prec_nnmf_cos(i) = getPrecision(indices_nnmf_cos(1:n(m)),
```

```
            querydata.relevantdocs{i});
94          prec_lsi_cos(i) = getPrecision(indices_lsi_cos(1:n(m)),
            querydata.relevantdocs{i});
95
96          recall(i) = getRecall(indices(1:n(m)),querydata.relevantdocs{i
            });
97          recall_svd(i) = getRecall(indices_svd(1:n(m)),querydata.
            relevantdocs{i});
98          recall_nnmf(i) = getRecall(indices_nnmf(1:n(m)),querydata.
            relevantdocs{i});
99          recall_lsi(i) = getRecall(indices_lsi(1:n(m)),querydata.
            relevantdocs{i});
100
101         recall_cos(i) = getRecall(indices_cos(1:n(m)),querydata.
            relevantdocs{i});
102         recall_svd_cos(i) = getRecall(indices_svd_cos(1:n(m)),
            querydata.relevantdocs{i});
103         recall_nnmf_cos(i) = getRecall(indices_nnmf_cos(1:n(m)),
            querydata.relevantdocs{i});
104         recall_lsi_cos(i) = getRecall(indices_lsi_cos(1:n(m)),
            querydata.relevantdocs{i});
105     end
106
107     avg_recall(j,m,1) = mean(recall);
108     avg_recall(j,m,2) = mean(recall_svd);
109     avg_recall(j,m,3) = mean(recall_nnmf);
110     avg_recall(j,m,4) = mean(recall_lsi);
111
112     avg_recall_cos(j,m,1) = mean(recall_cos);
113     avg_recall_cos(j,m,2) = mean(recall_svd_cos);
114     avg_recall_cos(j,m,3) = mean(recall_nnmf_cos);
115     avg_recall_cos(j,m,4) = mean(recall_lsi_cos);
116
117     avg_prec(j,m,1) = mean(prec);
118     avg_prec(j,m,2) = mean(prec_svd);
119     avg_prec(j,m,3) = mean(prec_nnmf);
120     avg_prec(j,m,4) = mean(prec_lsi);
121
122     avg_prec_cos(j,m,1) = mean(prec_cos);
123     avg_prec_cos(j,m,2) = mean(prec_svd_cos);
124     avg_prec_cos(j,m,3) = mean(prec_nnmf_cos);
125     avg_prec_cos(j,m,4) = mean(prec_lsi_cos);
126
127     %{
128     fprintf("--------------------------------\n");
129     fprintf("n = " + string(n(m)) + "\n");
130     fprintf("dot product\n" );
131     fprintf("average recall: %f\n", avg_recall(j,m,1));
132     fprintf("average recall svd: %f\n", avg_recall(j,m,2));
133     fprintf("average recall nnmf: %f\n", avg_recall(j,m,3));
134     fprintf("average recall lsi: %f\n\n", avg_recall(j,m,4));
135
```

```
136             fprintf("average prec: %f\n", avg_recall_cos(j,m,1));
137             fprintf("average prec svd: %f\n", avg_recall_cos(j,m,2));
138             fprintf("average prec nnmf: %f\n", avg_recall_cos(j,m,3));
139             fprintf("average prec lsi: %f\n\n", avg_recall_cos(j,m,4));
140 
141             fprintf("cos angle\n" );
142             fprintf("average recall: %f\n", avg_prec(j,m,1));
143             fprintf("average recall svd: %f\n", avg_prec(j,m,2));
144             fprintf("average recall nnmf: %f\n", avg_prec(j,m,3));
145             fprintf("average recall lsi: %f\n\n", avg_prec(j,m,4));
146 
147             fprintf("average prec: %f\n", avg_prec_cos(j,m,1));
148             fprintf("average prec svd: %f\n", avg_prec_cos(j,m,2));
149             fprintf("average prec nnmf: %f\n", avg_prec_cos(j,m,3));
150             fprintf("average prec lsi: %f\n\n", avg_prec_cos(j,m,4));
151            %}
152         end
153     end
154     diary off;
155 
156     plottingrank;
157     %plottingn;
158 end
159 
160 function [termlist,querydata] = modify_termlist_querydata(termlist_,querydata_
    )
161     %Convert termlist_.termlist from an array of structs to an array of
        strings.
162     nterms = termlist_.nterms;
163     termlist = strings(1, nterms);
164     for i = 1:nterms
165         termlist(i) = termlist_.termlist(i).term;
166     end
167 
168     %Convert querydata,query(i).words from an array of cells to an array of
        strings.
169     %Save the arrays of relavant doucments in the new structure as well.
170     nqueries = querydata_.nqueries;
171     querydata = struct('words',{cell(nqueries,1)},'relevantdocs',{cell(
        nqueries,1)});
172 
173     for i = 1:nqueries
174         querydata.words{i} = celltostring(querydata_.query(i).words);
175         querydata.relevantdocs{i} = querydata_.query(i).relevantdoc;
176     end
177 
178 end
```

Plotting

```
1 clf
```

```
subplot(2,2,1);
hold on
plot(k,avg_recall(:,4,1),'.-');
plot(k,avg_recall(:,4,2),'.-');
plot(k,avg_recall(:,4,3),'.-');
plot(k,avg_recall(:,4,4),'.-');
xlabel('k'); ylabel('recall');
legend({'vector space', 'SVD', 'NNMF', 'low dim'},'Location','southeast');
title('Recall vs Rank');
hold off

subplot(2,2,2);
hold on
plot(k,avg_recall_cos(:,4,1),'.-');
plot(k,avg_recall_cos(:,4,2),'.-');
plot(k,avg_recall_cos(:,4,3),'.-');
plot(k,avg_recall_cos(:,4,4),'.-');
xlabel('k'); ylabel('recall');
legend({'vector space', 'SVD', 'NNMF', 'low dim'},'Location','southeast');
title('Recall vs Rank (cos angle)');
hold off


subplot(2,2,3);
hold on
plot(k,avg_prec(:,4,1),'.-');
plot(k,avg_prec(:,4,2),'.-');
plot(k,avg_prec(:,4,3),'.-');
plot(k,avg_prec(:,4,4),'.-');
xlabel('k'); ylabel('prec');
legend({'vector space', 'SVD', 'NNMF', 'low dim'},'Location','southeast');
title('Prec vs Rank');
hold off

subplot(2,2,4);
hold on
plot(k,avg_prec_cos(:,4,1),'.-');
plot(k,avg_prec_cos(:,4,2),'.-');
plot(k,avg_prec_cos(:,4,3),'.-');
plot(k,avg_prec_cos(:,4,4),'.-');
xlabel('k'); ylabel('prec');
legend({'vector space', 'SVD', 'NNMF', 'low dim'},'Location','southeast');
title('Prec vs Rank (cos angle)');
hold off
```

For preprocessing the data:

```
function string_array = celltostring(cell_array)
%This function converts an array of cells to an array of strings.
    n = length(cell_array);
    string_array = strings(n,1);
```

```
    for i = 1:n
        string_array(i) =  cell_array{i};
    end
end
```

References

[1] A.B. Manwar, H.S. Mahalle, A Vector Space Model for Information Retrieval: A Matlab Approach, Indian Journal of Computer Science and Engineering, 3(2012), No.2, 222-229

[2] Dianne P. O'Leary, What's the Score? Matrices, Documents, and Queries, SIAM Press, Philadelphia, 2009

[3] Michael W. Berry, Zlatko Drmac, Elizabeth R. Jessup, Matrices, Vector Spaces, and Information Retrieval, Siam Review, 41(1999), No.2, 335-362

[4] Haesun Park, Moongu Jeon, J. Ben Rosen, Lower Dimensional Representation of Text Data based on Centroids and Least Squares, NSF, (2001)

[5] Patrik O. Hoyer, Non-Negative Matrix Factorization With Sparseness Constraints, Journal of Machine Learning Research, 5(2014), 1457–1469