Implementing Information Retrieval in MATLAB
MACM 409 \MATH 709

Xiao Rong (Michelle)Wu
Jarren Ralf

11 October 2019

Contents

Section 1

Xtract AI

The field of machine learning and Artificial Intelligence (AI) is one that is developing quickly due to the huge application potential of the subject. As a result, a large job market surrounds it, including an exemplary tech hug in metro Vancouver. One such company in this line of work is called Xtract AI, situated in Vancouver, BC. Xtract contributes to a large variety of industrial fields, including defense and security, health care, telecommunications, environment and transportation. Solutions to various problems within these fields has Xtract analyse and interpret video data, extract data from documents, do geospatial analysis, and image analysis.

The annual revenue of Xtract is not provided explicitly but is estimated to be $3 million dollars by the website Crunchbase [1]. There are eleven employees according to their own biography page on their website.

As listed above, one of the most interesting applications that Xtract focuses on is document data extraction and analysis. They have carved their procedure down to the following few components: classification of millions of documents, trained data set for key content, relevant results returned rapidly, and stunning user interface with intuitive design. [3].

The application that Xtract usually applies this particular task to, is searching and retrieving data in medical journals.

In addition, to avoid the missing of key documents, document analysis is necessary. Interpreting or analyzing the semantic meaning of key words helps to reduce risk of missing information. By doing that, Xtract provides more comprehensive resources to its clients. To support their claim, they state the following,"a public health authority was trying to make it easier for physicians and researchers to find critical data pertaining to medical studies. Due to the disparity of digital data sets, lack of consistency across document types and naming conventions, it could take researchers up to a year to discover and extract all the relevant publications and insights for a given field of study." This process is also referred to as intelligent information retrieval.

Section 2

Introduction of the Problem

Information is the most powerful commodity that human beings possess. Groups of people who have information that others don't have distinct advantage in many scenarios. The act of sharing information is naturally derived from our ancestral lineages where trading was prevalent. The act of spreading information among the species has experienced major upsurge with the development of technology. Perhaps the first example of this would be the act of writing. Naturally, the printing press provided a large boost to the circulation of information within large communities. Currently, access to information has exploded with the continued development of the internet. Distribution is at an all time high, and is continuing to increase. So information is spread around the globe with ease. Businesses who thrive on the development and exchange of information now face a problem unique to our time as human beings. How do we retrieve this abundance of information in the best possible way?

Information retrieval is a general enough topic, but for a company like Xtract AI, their focus is for medical information. The task we look to implement is finding the most relevant document that suits out users needs. With this set as our goal, we begin by giving a quick background on information retrieval and a brief literature review.

2.1 Information Retrieval

Broadly speaking, data extraction is a procedure to access information of data from various sources. Imagine attempting to access data in the English language while deal with the phenomena of multiple means for the identical word. How would you be able to properly map the input information to the correct output source? There is also the factor of synonym words that may be spelled completely distinct from the searching string, however the meaning overlaps, with others. Just to share another example, naming conventions for the same topic could be different based on department or even region. These are a few examples of the challenges someone implementing intelligent information retrieval may face.

The main data science questions that are inherent to this problem are, first, that of working with a large sparse matrix and determining the most important entries in that matrix. If this matrix represents a set of words which map to a particular document, based on the search criteria, some of the information in the matrix becomes more valuable than others.

The domain we will choose to embed this problem in, is that of searching through medical documents to find relevant articles based on the search words. Providing comprehensive information

to health professionals would assist in supporting them to give an accurate treatment, diagnoses, or recommendation. Therefore, spending time on investigating the applications to search medical journal is necessary. The question that could be asked of us is "how to find medical documents based on the search words?" A simple model will be the focus of our project in the beginning. Since intelligent information retrieval is the forefront of today's data extraction and analysis, we will drop the 'intelligent' and build a simple searching system in phase 1. For example, a first version of the code may simply search for exact sting matches, instead of interpreting the meaning of the words.

2.2 Literary Review

2.2.1 Text classification by aggregation of svd Eigenvectors

In this paper, the author categorizes documents by their reading easiness. They apply the vector space model to the documents. They compared the following three approaches, using the SVD, aggregated SVD and the Flesh Reading Ease index for dimensional reduction, which categorize documents by readability easiness. Based on the information the text file provided, they transform words in documents to a term-document matrix. And they rank the matrix into three categories, easy, medium and advanced.

The paper first mentions the classic SVD method and applies the decomposition technique to a $m \times n$ matrix A and it can decomposed into three matrices: an $m \times m$ orthogonal matrix U, a $m \times n$ diagonal matrix Σ and the transpose of an n×n orthogonal matrix V.The following is the SVD formula

$$A = U\Sigma V^T.$$

The paper also mentions that recomputing the SVD of a new term-document matrix or folding-in the new terms and documents are two alternative method to deal with adding terms or documents to the existing matrix.

Furthermore, they applied cosine similarity formula to the matrix U and compared it to every row of matrix U. To be efficient, using SVD for dimensional reduction is necessary. By proceeding dimensional reduction, they remove noise word and keep 80 percent information of the term document matrix.

Finally, the author compared the performance of four methods in classifying text documents by readability easiness. The aggregated SVD had the best performance and the Flesh Reading Ease does worst performance.

2.2.2 Using Linear Algebra For Intelligent Information Retrical

Same as last paper, the matrix A can factor into the product of 3 matrices by using the singular value decomposition (SVD) such as $A = U\Sigma V^T$. Since the full SVD requires large memory, the author also briefly introduces low rank approximation. They use k-largest singular triplets to approximate the original term-document matrix by A_k since

$$A_k = \sum_k U_k \Sigma_k V_k^T.$$

They compared the query vector q with existing document vectors in matrix A or U, and the documents ranked by their similarity to the query after applying cos similarity. A way to im-

prove retrieval performance is using the appropriate term weighting. This implies that apply some functions to the frequency of occurrence of a term in the term-document matrix.

2.3 Using Linear Algebra to Extract Data from Documents

In this section, we apply linear algebra to find information we requested. Usually, information is retrieved by literally matching terms in documents with a query. This requires building a large sparse matrix. For large examples, this could be very computationally expensive. Using a singular value decomposition is a viable way to create a good approximation to our sparse matrix system.

2.3.1 Background information

We are going to discuss the background information we used. The idea of matching words from multiple documents is inherently complex due to the sophistication of human language. For example in the English dictionary, there are many words that map to multiple definitions. This provides a challenge when attempting to match text and meaning. One way of dealing with this is proposed by Berry *et al*, which is using Latent Semantic Indexing (LSI) [2]. LSI is an information retrieval method that uses Singular Value Decomposition (SVD) to identify patterns between terms and concepts in a set of texts. So in order to implement LSI, a matrix must be construct whose elements are each of the terms in the various documents. For example denote

$$A = [a_{ij}]$$

where a_{ij} denotes the frequency in which term i occurs in document j. This means that A is likely to be a very sparse matrix because most documents do not contain every word we are searching for. After creating a term by document matrix,we can compute the SVD of this matrix, create the database of singular values and vectors for retrieval, and match user queries to documents. In addition, constructing a dictionary vector which containing all the words in documents are helpful. The singular value decomposition (SVD) will estimate the structure in word usage across documents.

We also need to create a $m \times 1$ query vector q. Inside the vector q, the position of the number 1 in an entry represents the term occurs once in the query vector. For purposes of information retrieval, a user's query can be represented as a vector q in k-dimensional or n-dimensional space. A query (like a document) is a set of words. The length of the query vector q will be k or n. The number in the query vector represents the occurrence of a specific word in the dictionary. For example, the user query can be represented by

$$q = q^T U_k \sigma_k^{-1}$$

The query vector will used to compared with all existing document vectors in the matrix A and and find most relevant documents.For example, the most relevant to query terms could be computed by $y = A^T q$, but actually we approach the result in a different way.

Furthermore, we will use measure the cosine between the query vector and document vector to find the similarity between them since there may be documents that are more relevant than another. Using the value of cosine as a measure tells us what's the better match. For example, the

closer $\cos\theta = 1$, then the better the match of the query vector, q, and the i^{th} document [2]. The following computation may need and p_j given by

$$p_j = \frac{1}{\|a_j\|} a_j$$

where $j = 1, 2, \ldots$ are the indexes representing the columns of a matrix P formed by each p_j. Observe the formula for the cosine of the angle between vectors q and p_i, given by

$$y_i = p_i^T q = \cos\theta_i$$

where the i^{th} entry of y_j represents the number of query terms present in the document i [2].

By applying the SVD, documents that are most relevant to a particular query will return to the user. The following is the description of singular value decomposition:

Theorem 3.1. If $A \in M^{m\times n}$ has rank r, then it can factored in the form $A = U\Sigma V^T$ where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and $\Sigma = \sigma_{i,j}$ is an $m \times n$ matrix whose off-diagonal entries are all 0, and

$$\sigma_{1,1} \geq \sigma_{2,2} \geq \cdots \geq \sigma_{r,r} > \sigma_{r+1,r+1} = \cdots = \sigma_{q,q} = 0$$

where $q = \min\{m, n\}$.

In order to increase efficiency of our algorithm, we apply low rank approximation to the matrix A, and use k largest singular values to approximates the original term-by-document matrix A by A_k. The Eckart and Young theorem illustrate the truth.

Theorem 2.2 [Eckart and Young]Let the SVD of a $m\times$n matrix A with r = rank(A) $\leq$ p=min(m,n) and define A_k.

$$A_i = \sum_{n=k}^{i=1} u_i * \sigma_i * v_i^T$$

then

$$min||A - B||_F^2 = ||A - A_k||_F^2 = \sigma_{k+1}^2 + \sigma_{k+2}^2 + \cdots + \sigma_p^2$$

Applying SVD to our matrix A_k, the document most relevant to a particular information returns to the user again.

The last key idea we will share is the one used to get rid of "noise." First of all, what is noise in our problem? An example of noise would be the kinds of words that show up in every document, for example, 'a', 'the', 'to', 'from' etc. It is a good idea to remove noisy data, which in turn will help to return the desired result efficiently.

We also need to apply tf-idf weight to both our matrix and query vector since it makes our final result accurate.Tf-idf represents term frequency-inverse document frequency, and the tf-idf weight is often used in information retrieval. The following are definitions of tf-idf weight in wekipedia:

The tf–idf is the product of two statistics, term frequency and inverse document frequency. There are various ways for determining the exact values of both statistics.

The formula that aims to define the importance of a keyword or phrase within a document or a web page.

Recommended tf–idf weighting schemes

weighting scheme	document term weight	query term weight
1	$f_{t,d} \cdot \log \frac{N}{n_t}$	$\left(0.5 + 0.5 \frac{f_{t,q}}{\max_t f_{t,q}}\right) \cdot \log \frac{N}{n_t}$
2	$1 + \log f_{t,d}$	$\log\left(1 + \frac{N}{n_t}\right)$
3	$(1 + \log f_{t,d}) \cdot \log \frac{N}{n_t}$	$(1 + \log f_{t,q}) \cdot \log \frac{N}{n_t}$

2.3.2 The example of searching mental health documents

We talk about an example by simply searching a word for exact string matches. As we mentioned before, we have to generate a term-document matrix, a dictionary vector and a query vector. The dictionary vector contains all the unique words which are in any of the documents. Every word in the dictionary should be unique. The length of each row of term-document matrix is same as the length of dictionary. For example, if the entry a_{ij} in term-document matrix A equals to 3, this represents that the term j occurs 3 times in i_{th} document. In the following, we apply the vector space model to our simple example:

D1:Behavioral Health Mental Health
D2:Personality and Behavior Changes
D3:Belief in supernatural causes of mental illness
D4:Genetic bases of mental illness
D5:Social capital and mental illness
D6:Biogenetic explanations and public acceptance of mental illness
D7:The effect of exercise on clinical depression
D8:Mental Health Conditions
D9:Mental health and work
The dictionary is the vector:

Dictionary = [“behavioral” “health” “mental” “personality” “and” “behavior” “changes” “belief” “in” “supernatural” “causes” “of” “illness” “genetic” “bases” “social” “capital” “conditions” “biogenetic” “explanations” “public” “acceptance” “the” “effect” “exercise” “on” “clinical” “depression” “work”]

Since dictionary is a 1×29 vector, the each row of the document-term matrix should be 1×29. For example, in the document 1, we counted the frequency of a specific word, and located frequency of this word in the column of the term-document matrix which same the column the word located at dictionary. We repeats this procedure until we transform all the words to a frequency matrix. The updated entries of each row of matrix will be like following:

D1=[1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
D2=[0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
D3=[0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Documents	Similarity with document
D1	0.0069
D2	0
D3	0.4108
D4	0.1249
D5	0.0653
D6	0.0898
D7	0.3034
D8	0.0081
D9	0.0076

Table 1. Similarity of the query vector with the nine documents (before removing noise)

D4=[0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
D5=[0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0]
D6=[0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0]
D7=[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0]
D8=[0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
D9=[0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

We observe entries of each row in our document-term matrix, the frequency value are very small. The frequency of some terms, like “in”, “of”, “the” and “on” are the big portion of our matrix. However, these term are not useful for searching medical documents. But, in this example, we will keep this words, observe how does the existence of these words affect our result and apply the tf-idf weighting to our frequency matrix. After that, we would apply low rank approximation and singular value decomposition to remove these information to our example.

Next, we create a query vector, for example we like to search the causes of mental illness then our query vector q would be :

query=[“the” “causes” “of” “mental” “illness” “”];

Then, we proceed the same procedure as we did for term-document matrix to our query vector, then frequency query will like the following:

query=[0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

Lastly, we compared the query vector with the 9 documents(D1-D9) from the term-document matrix. More specifically we compared it to every row of our matrix A by using cosine similarity. The first vector (row) of matrix A corresponds to document D1, the second to document D2, etc. The calculated cosine similarities are shown in table 1.

Based on Table 1, the query vector is most similar with D3 and D7.

Section 3

Algorithm Performance

We ran our code for the example we did by hand in section 2.3.2, and we were able to reproduce the exact same results.

3.1 Stability

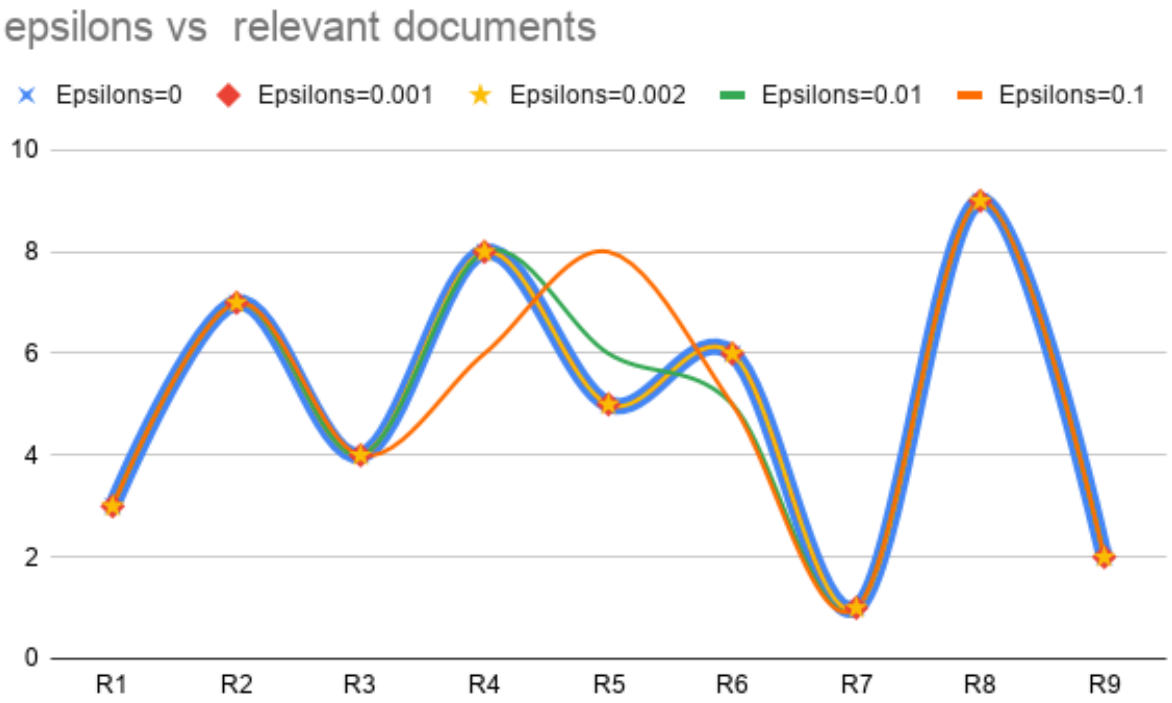

Figure 3.1: plot of relevant documents vs. epsilon

The above is the table of epsilons versa relevant documents for our small example. We apply the rank-5 approximation to this 9×29 document-term-matrix, the relevant documents in descending

order would be document 3, document 7, document 4, document 8, document 5, document 6 document 1, document 9 and document 2. Since we would like to know the stability of our example, we perturb the query vector. From the table, when epsilon ≤ 0.0002, the order of relevant documents does not change . As epsilon increases to 0.01, the order of 5th most relevant matrix switch with order of 6th most relevant matrix. When the epsilon equal to 0.1, three documents change order. This implies our algorithm is stable since the order of relevant documents does not change much with the small epsilon.

	R1	R2	R3	R4	R5	R6	R7	R8	R9
Epsilons=0	3	7	4	8	5	6	1	9	2
Epsilons=0.001	3	7	4	8	5	6	1	9	2
Epsilons=0.002	3	7	4	8	5	6	1	9	2
Epsilons=0.01	3	7	4	8	6	5	1	9	2
Epsilons=0.1	3	7	4	6	8	5	1	9	2

Figure 3.2: Table 2: The relevant document with perturbation for the rank 5 approximation case

3.2 Efficiency

MATLAB allows for some very efficient operations when it comes to dealing with matrix systems. In typical algorithms, traversing an array would require a double `for`-loop, but within matlab, sometimes operations that are suppose to happen entry wise, can be computed simultaneously. This process is referred to by vectorization.

We did a fairly good job in vectorizing many of our commands. For example, you can see in Listing A.4, A.6, and A.7, that no loops were used in the creation of these arrays. There were a couple more places where we could have incorporated this concept, for example with the calculation of the cosine similarities. But in general, the vectorization we did include certainly helped our code run faster in terms of efficiency. Below are some examples of numerical experimental results to show this.

In Figure 3.3, the plot of the elapsed time of the full rank SVD vs rank-5 SVD. We apply the rank approximation to this 9* 29 document-term-matrix, the elapsed time for rank 1 approximation to full rank SVD are 0.003453, 0.002175, 0.001262, 0.001280, 0.001811, 0.001294, 0.001253,0.001310 and 0.005755 seconds. Obviously, compare with rank-k approximation, the full rank SVD takes more time since the elapsed time of the majority of rank-k approximation only spends 1/3 of the elapsed time of the full rank svd for our small example. Thus, applying rank-k approximation to our matrix is useful to increase the efficiency of the algorithm.

Another way we developed some efficiency in our code was low storage cost. Within MATLAB, there are many ways to deal with sparse matrices. The idea here is that the system only need to know the position and the value of the nonzero entries. The code used to create these matrices comes from the command `sparse`. The query vector is our first example of creating a sparse object. Naturally, we further exploited this function in creating the entire *td* matrix as well. One improvement that we could have added was to use the sparse SVD call `svds`. We had some challenges with this implementation, but in future versions of this program those bugs would be worked out. Instead, in the interest of time, we converted out system to a full matrix to compute the SVD.

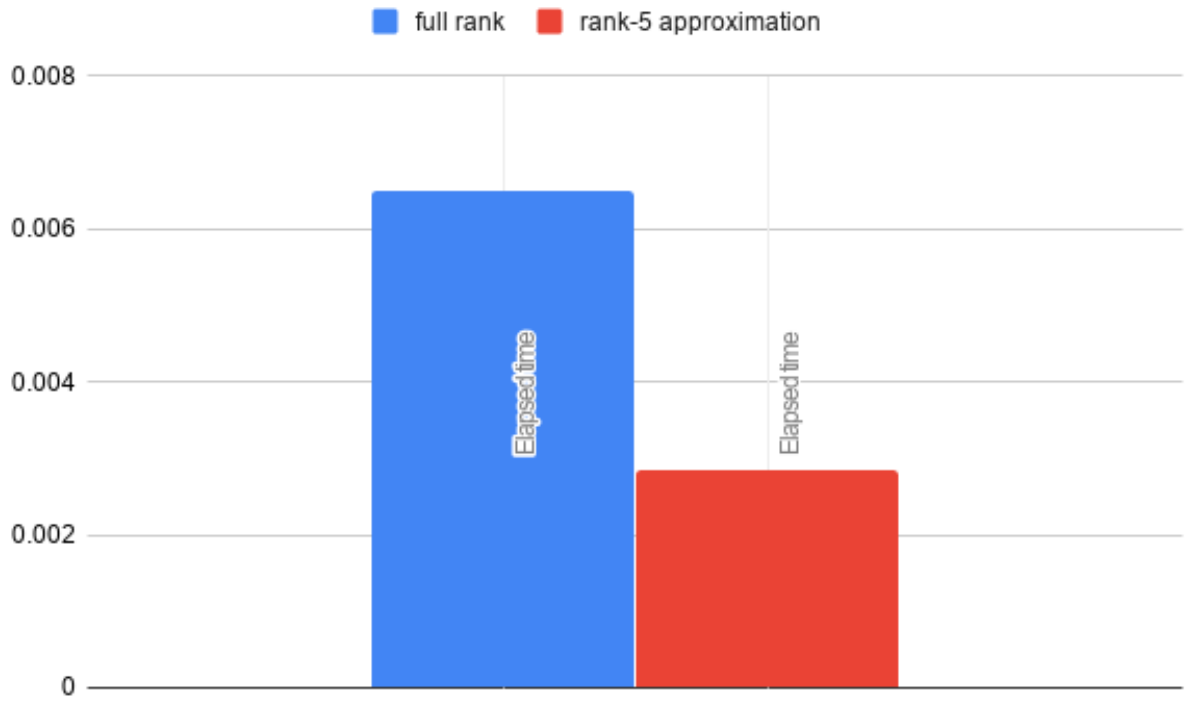

Figure 3.3: plot of the elapsed time of the full rank SVD vs rank-5 SVD

3.3 Accuracy

k=9	k=8	k=7	k=6	k=5	k=4	k=3	k=2	k=1
0.0069	0.0081	0.0066	0.0512	0.0826	0.1633	0.2236	0.2959	0.3442
0	0	-0.0002	-0.0002	-0.001	0.0019	-0.0103	0.2876	0.3442
0.4108	0.4108	0.4108	0.4108	0.4114	0.416	0.416	0.3212	0.3442
0.1249	0.1249	0.125	0.125	0.1716	0.4007	0.4022	0.3634	0.3442
0.0653	0.0653	0.0651	0.0659	0.0917	0.1902	0.2277	0.2979	0.3442
0.0898	0.0898	0.0897	0.0897	0.0898	0.0942	0.0954	0.3113	0.3442
0.3034	0.3034	0.3034	0.3034	0.3036	0.3038	0.3038	0.3073	0.3442
0.0081	0.012	0.0108	0.074	0.1065	0.22	0.2629	0.2987	0.3442
0.0076	0.0077	0.0205	0.0488	0.0682	0.0629	0.0873	0.2959	0.3442

Figure 3.4: Table 3: The table of similarity values with k-rank approximation

The above is the table of similarity values after applying the k-rank approximation for our example. Since the size of our document-term-matrix is 9×29, the maximum rank of our matrix is nine. After our demonstration, we figure out that the rank of the matrix is nine. When we apply full rank SVD to the document-term-matrix, we get five most relevant documents in descending order which are document 3, document 7, document 4,document 6 and document 5. From our observation, when $k \geq 6$, the similarity value after proceeding k-rank approximation are quite similar as the result after proceeding full rank SVD. When $k = 5$, there are only 3 most relevant

documents return in the correct order. Therefore, if we want our algorithm return at least 40% of correct information back to user, we are better to use the $k \geq 6$. The figure 1 shows the trends of similarity values with respect w to k-rank approximation.

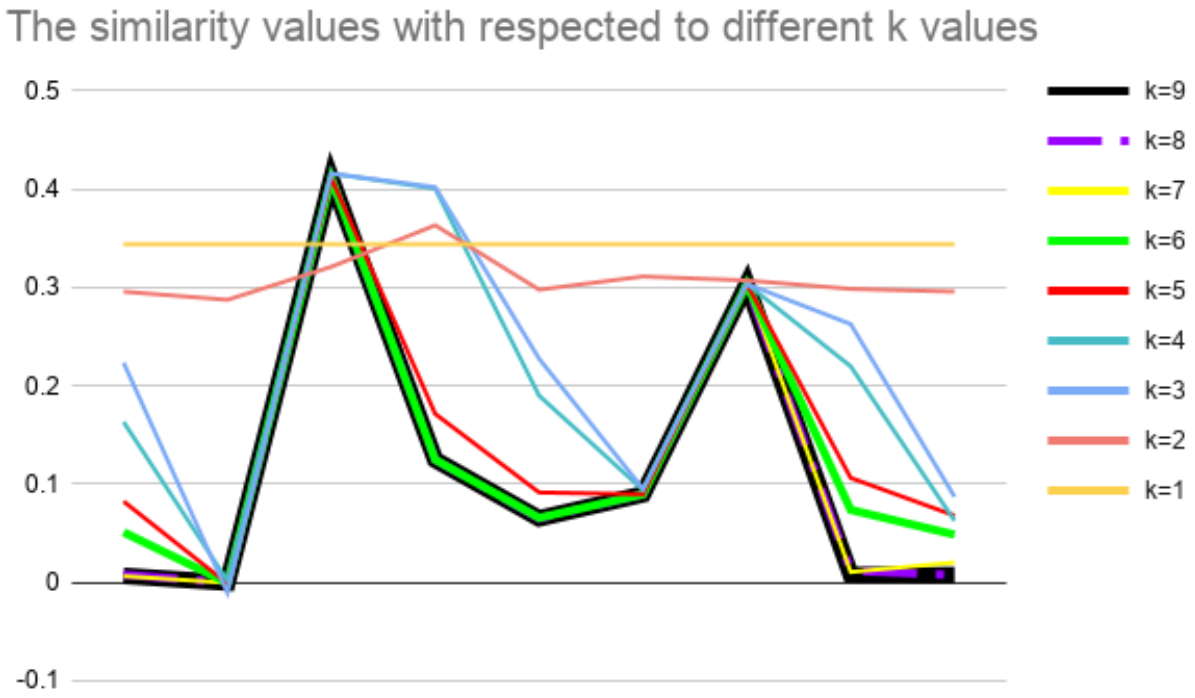

Figure 3.5: plot of similarity values of K-rank approximation

The figure 1 clear shows that if we choose the rank 7 approximation to our document term matrix , we will get the general pattern which is similar as the full rank SVD. When $k \leq 6$, the k-rank approximation of the document term matrix becomes inaccurate.

Section 4

Summary

Based on a few small examples in the literature, we successfully implemented an information retrieval program. If the the user wants to find relevant medical data within a set of documents, then we can return an ordered list of the most relevant documents. This has the potential to be a very computationally expensive procedure for large systems, so our way of dealing with this was using the SVD to obtain a rank k-approximation to our term document matrix. This gave us some increased speeds with some decreased accuracy. Decreased accuracy means the order of most relevant documents change. When k become smaller, our result become worse and worse. Better efficiency means lower computation cost because we compute the singular values.

If we had more time to improve the code, we would work to vectorize more or the operations, as well as fully exploit the use of sparse matrices. We had a larger example that we did not get around to testing either. Our code also had the ability to remove "noisy words," or as we label them "common words." Our code reads in a second document of the most common words and can delete these from the dictionary and documents. We would have liked to run numerical experiments to see the effect of this.

Appendix A

Matlab Code

Listing A.1: Retrieving words from a txt. document

```
function terms = getWords(txtDocument, removeCommonWords)
% This function receives as input txt documents, and and boolean condition for deleting
% common English words. It will return a list of words as a column string array.
%
%       txtDocuments = A vector of strings or a character array of document names
%  removeCommonWords = A boolean that controls whether or not to remove common words
%
% @author Jarren Ralf

NUM_CHAR = 3204;      % The number of charcters in the common_words.txt document
MAX_NUM_CHAR = 5000; % The max number of charcters to read into the matrix (Arbitrary)
fileID = fopen(txtDocument, 'r', 'n', 'UTF-8'); % Read in .txt file as a character array
termsMTX = fread(fileID, [1 MAX_NUM_CHAR], '*char');
termsMTX = string(termsMTX);      % Convert character array to string array
termsMTX = splitlines(termsMTX); % Split strings at newline characters
TF = (termsMTX == "");            % Find blank strings of text ...
termsMTX(TF) = [];                % ... and remove them
% Replace unwanted charaters with spaces i.e. punctuation and numbers
p = ["." "?" "!" "," ";" ":" "%" "(" ")" "[" "]" "=" "+" "-" "?" "/" "'" 0:9];
termsMTX = replace(termsMTX, p, " ");
% Strip leading and trailing blank space characters from each element
termsMTX = strip(termsMTX);
terms = strings(0); % Strip the strings to individual words and store in column vector
for i = 1:length(termsMTX)
    terms = [terms ; split(termsMTX(i))];
end
% Remove the common words from the terms vector if true (chosen by the user)
if removeCommonWords == true
fileID_CW = fopen('common_words.txt', 'r', 'n', 'UTF-8'); % Read in .txt file
commonWords = fread(fileID_CW, [1 NUM_CHAR], '*char');    % as a character array
commonWords = string(commonWords);      % Convert character array to string array
commonWords = splitlines(commonWords); % Split strings at newline characters
% Delete the commom words from our terms vector **Notice that duplicates are kept
terms = terms(~ismember(terms, commonWords));
end
fclose('all');
```

Listing A.2: Creating the Dictionary

```
function dictionary = createDictionary(txtDocuments, removeCommonWords)
% This function accepts as input the txt documents, and boolean condition for deleting
% common English words. The output is the dictionary as a vector of strings representing all
% words used across each of the documents.
%
%       txtDocuments = A vector of strings or a character array of document names
%  removeCommonWords = A boolean that controls whether or not to remove common words
%
% @author Jarren Ralf

d = length(txtDocuments); % The number of documents
for i = 1:d
    % Return a column vector of words that where contained in the chosen document
    terms = getWords(txtDocuments(i), removeCommonWords);
    if i == 1                                % For the first document in the list
        currentDictionary = unique(terms);  % Alphabetizes and deletes repetitive words
        if d == 1                            % If there is only 1 document ...
            dictionary = currentDictionary; % ... then return this sorted list ...
            break;                           % ... by breaking the for loop
        end
    else
        concatenatedList = [currentDictionary; terms];
        currentDictionary = unique(concatenatedList);
    end
end
dictionary = currentDictionary;
```

Listing A.3: Building the numerical query vector

```
function q = getQueryVector(query, dictionary)
% This function receives as input a query vector and dictionary vector of strings. The
% output is a numerical query vector which places a 1 the ith position corresponding to the
% ith word in the dictionary.
%
%              query = A vector of strings that represents the question asked
%         dictionary = A vector of strings (all words used across each of the documents)
%
% @author Jarren Ralf

% i is the index in the dictionary which each string of the query vector belongs
[~,i] = intersect(dictionary, query);
j = ones(length(i), 1);                 % Column: 1
v = ones(length(i), 1);                 % Entry:  1
q = sparse(i, j, v);                    % Build sparse query vector
q(end + 1:length(dictionary), 1) = 0; % Append zeros so it is the correct length
```

Listing A.4: Calculate the idf values

```
function idfValues = idfvalue(termDocumentMatrix)
% This function accepts as input the Term Document Matrix. The output is a vector of the
% idfValues corresponding to each term.
%
%     termDocumentMatrix = numerical term-document (td) matrix
%
% @author Michelle Wu amd Jarren Ralf

[d, t] = size(termDocumentMatrix); % The number of documents, d, and the number of terms, t
totalFrequency = sum(termDocumentMatrix); % Calculate the total frequency for each term
idfValues = log2(d./(totalFrequency));    % Calculate the idf values
```

Listing A.5: Creating the Term-Document (td) Matrix

```
function td = createTermDocMtx(dictionary, txtDocuments, removeCommonWords)
% This function accepts as input the dictionary, txt documents, and boolean condition for
% deleting common English words. The output is a numerical term-document (td) matrix where
% td_{ij} is the frequency of the jth term occuring in the ith document.
%
%          dictionary = A vector of strings (all words used across each of the documents)
%       txtDocuments = A vector of strings or a character array of document names
%  removeCommonWords = A boolean that controls whether or not to remove common words
%
% @author Jarren Ralf

d  = length(txtDocuments); % The number of documents
t  = length(dictionary);   % The number of terms
td = sparse(d, t);         % Preallocate the sparse td matrix
for i = 1:d                % Loop through all of the documents
    terms = getWords(txtDocuments(i), removeCommonWords);
    % Place a set of ones in row i to represent the occurences of term j
    td(i, :) = getQueryVector(terms, dictionary)';
       uniqueTerms = unique(terms);       % Retrieve the unique terms
    numUniqueTerms = length(uniqueTerms); % The number of unique terms
    freq = ones(1, numUniqueTerms);       % Preallocate the frequency vector
    for k = 1:numUniqueTerms
        freq(k) = sum(ismember(terms, uniqueTerms(k, :)));  % Compute the frequency
    end
    counter = 1;                                  % Initialize a counter
    numRepetitiveWords = sum(freq(:) > 1);        % Total number of repetitive words
    freq_max  = max(freq);                        % The maximum frequency
    freq_list = strings(numRepetitiveWords, 1); % The list of repetitive words
    freq_vals =    ones(numRepetitiveWords, 1); % The list of frequencies
    % Store only the repetitive terms and their corresponding frequencies
    for j = 1:numUniqueTerms
        if freq(j) > 1
            freq_list(counter) = uniqueTerms(j);
            freq_vals(counter) = freq(j);
            counter = counter + 1;
        end
    end
    % Update the frequencies of the occurences of words in row i of the td matrix
    for f = 1:freq_max - 1           % These are the cases where a term occurs > 1 times
        [r, c] = find(freq_vals > f); % f is the frequency of a word occuring
        td(i, :) = td(i, :) + getQueryVector(freq_list(r), dictionary)'; % Update row
    end
end
```

Listing A.6: Apply the idf Values to the Term-Document (td) Matrix

```
function tfScoreMatrix = tfIdf(termDocumentMatrix)
% This function accepts as input the term-document Matrix. The output is a matrix of the
% idfValues in their corresponding position in the td matrix.
%
%     termDocumentMatrix = numerical term-document matrix
%
% @author Michelle Wu and Jarren Ralf

idfValues = idfvalue(termDocumentMatrix);
tfScoreMatrix = termDocumentMatrix.*idfValues;
```

Listing A.7: Apply the idf Values to the Query vector

```
function tfidfQuery = tfidfQuery(termDocumentMatrix, query)
% This function accepts as input the term-document matrix and a query vector. The output is
% the Query vector with the idf values applied to it.
%
%     termDocumentMatrix = Numerical term-document matrix
%                  query = Numerical query vector
%
% @author Michelle Wu and Jarren Ralf

idfValues = idfvalue(termDocumentMatrix);  % Apply the idf values to the td matrix
tfidfQuery = query.*idfValues'/max(query); % Apply the idf values to the query vector
```

Listing A.8: Determine the cosine similarity values for each document

```
function cosSim = simmilarityValue(termDocumentMatrix, query)
% This function accepts as input the term-document matrix and a query vector. The output is
% a vecotr of the cosine similarity values.
%
%     termDocumentMatrix = Numerical term-document matrix
%                  query = Numerical query vector
%
% @author Michell Wu

[d, t] = size(termDocumentMatrix); % The number of documents, d, and the number of terms, t
cosSim = zeros(d, 1);
tfScoreMatrix = tfIdf(termDocumentMatrix);       % final matrix without SVD
tfQuery = tfidfQuery(termDocumentMatrix, query); % the final query vector
documentLength = vecnorm(tfScoreMatrix');
queryLength = norm(tfQuery);
for i = 1:d
    total = 0;
    for  j = 1:t
        total = tfQuery(j)'*tfScoreMatrix(i, j); % The row total
    end
    cosSim(i) = total/(queryLength*documentLength(i)); % The cosine similarities
end
```

Listing A.9: Return the relevant documents to the user based on their query/question

```
function message = getDocuments(query, txtDocuments, removeCommonWords, numRelDocs)
% This function accepts as input the cosine simularities and the number of relevant
% documents the user wants returned. The output is the d number of relevant documents.
%
%              query = A vector of strings that represents the question asked
%       txtDocuments = A vector of strings or a character array of document names
%  removeCommonWords = A boolean that controls whether or not to remove common words
%         numRelDocs = The number of relevant documents the user wants returned
%
% @author Michelle Wu and Jarren Ralf

if nargin < 4       % If the user doesn't input the number of relevant articles ...
    numRelDocs = 0; % ... then return all of them, "mostRelevantDocument" will handle this
end
query = split(string(query));                          % Make the query a vector of strings
dictionary = createDictionary(txtDocuments, removeCommonWords);     % Create the dictionary
td = createTermDocMtx(dictionary, txtDocuments, removeCommonWords); % Build the td matrix
q = getQueryVector(query, dictionary);                 % Compute the numerical query vector
cosSim = simmilarityValue(td, q);                      % Calculate the cosine similarity
message = mostRelevantDocument(cosSim, numRelDocs); % Output the relevant documents
```

Listing A.10: Approximate the cosine similarity values for each document using the SVD

```
function cosSim = simmilarityValueSVD(termDocumentMatrix, query, k)
% This function accepts as input the term-document matrix and a query vector. The output is
% a vecotr of the cosine similarity values.
%
%     termDocumentMatrix = Numerical term-document matrix
%                  query = Numerical query vector
%
% @author Michell Wu

[d, t] = size(termDocumentMatrix); % The number of documents, d, and the number of terms, t
tfScoreMatrix = full(tfIdf(termDocumentMatrix));
[U, S, V] = svd(tfScoreMatrix);
tfScoreMatrixSVD = zeros(size(tfScoreMatrix));
for i = 1:k
    tfScoreMatrixSVD_i = S(i,i)*U(:,i)*V(:,i)'; % Apply the rank k approximation
    tfScoreMatrixSVD = tfScoreMatrixSVD + tfScoreMatrixSVD_i;
end
tfQuery = tfidfQuery(termDocumentMatrix, query); %the final query vector
documentLength = vecnorm(tfScoreMatrix');
queryLength = norm(tfQuery);
cosSim = zeros(d, 1);
for i = 1:d
    total = 0;
    for  j = 1:t
        total = total +tfQuery(j)*tfScoreMatrixSVD(i, j);
    end
    cosSim(i) = total/(queryLength*documentLength(i));
end
```

Listing A.11: Return the approximate relevant documents to the user based on their query/question

```
function message = getDocsApprox(query, txtDocuments, removeCommonWords, k, numRelDocs)
% This function accepts as input the cosine simularities and the number of relevant
% documents the user wants returned. The output is the d number of relevant documents.
%
%              query = A vector of strings that represents the question asked
%       txtDocuments = A vector of strings or a character array of document names
%  removeCommonWords = A boolean that controls whether or not to remove common words
%                  k = An integer that represents the rank-k approximation to the td matrix
%         numRelDocs = The number of relevant documents the user wants returned
%
% @author Michelle Wu and Jarren Ralf

if nargin < 5          % If the user doesn't input the number of relevant articles ...
    if nargin < 4
        k = ceil(length(txtDocuments)/2);
    end
    numRelDocs = 0; % ... then return all of them.
end

query = split(string(query)); % Make the query a vector of strings
dictionary = createDictionary(txtDocuments, removeCommonWords);     % Create the dictionary
td = createTermDocMtx(dictionary, txtDocuments, removeCommonWords); % Build the td matrix
q = getQueryVector(query, dictionary); % Set up the query vector
cosSim = simmilarityValueSVD(td, q, k); % Calculate the cosine similarity
message = mostRelevantDocument(cosSim, numRelDocs);
```

Listing A.12: The script to run our designed test case

```
clear; close all; clc;

% Choose the number of relevant documents you want return (default is all)
%numRelevantDocuments = 8;

% Choose the rank k approximation from k = 1:0
%k = 8;

% Choose your query
%query = ["the" "cause" "of" "mental" "illness"]; the cause of mental illness

% Or Type your own query
prompt = 'Please ask your question\n';
query = input(prompt, 's');

% Prepare the text documents for input
txtDocuments = strings(9, 1);
for i = 1:9
    txtDocuments(i) = strcat("TestCase/", num2str(i), ".txt");
end

removeCommonWords = false;

message1 = getDocuments(query, txtDocuments, removeCommonWords);
%message = getDocuments(query, txtDocuments, removeCommonWords, numRelevantDocuments);

disp('This is the approximation using the SVD.')
disp('The default rank is ceil(d/2) (half the number of documents.)')
message2 = getDocsApprox(query, txtDocuments, removeCommonWords);
%message2 = getDocsApprox(query, txtDocuments, removeCommonWords, k);
%message2 = getDocuments(query, txtDocuments, removeCommonWords, numRelevantDocuments);
```

Bibliography

[1] M. Arrington, *Crunchbase*. https://www.crunchbase.com/organization/xtract-technologies#section-overview. Accessed: 2019-10-08.

[2] M. W. Berry, S. T. Dumais, and G. W. O'Brien, *Using linear algebra for intelligent information retrieval*, SIAM review, 37 (1995), pp. 573–595.

[3] E. Holtham, *Xtract Technologies Inc.* https://xtract.ai/. Accessed: 2019-10-08.