Beckett Hyde, Ferin Von Reich
APPM 3310 Matrix Methods and Applications
July 10, 2022

Vector Space Model of Information Retrieval Project

Contents

1 Abstract

In this project, we implement a vector space model of information retrieval to index and search a corpus of text. In the vector space model, one generates an n-dimensional vector for each document where each component represents term frequency. Then, these vectors are made the columns of an $n \times d$ matrix, and querying is conducted by determining which of the columns are geometrically similar to a vector representing the query.

More specifically, we lay the mathematical foundations for the model and demonstrate it on a five-document example corpus. We then apply it to a set of pages from the University of Colorado-Boulder's website and test it on three carefully chosen test queries. Finally, we qualitatively find that the pages returned by the model are relevant and useful for each query and that the model performs well under both ideal and non-ideal conditions.

2 Attribution

Ferin wrote the sitemap generator for getting University of Colorado-Boulder URLs. Beckett wrote the scraper which takes those URLs and gets the text data for the creation of the database matrix. Both worked on the code which translates the text data into a dictionary, generates a matrix, performs SVD, and performs a query. Both contributed to the report.

3 Introduction

Information Retrieval from a large corpus of text is a persistent problem in modern computing. We have access to more useful information today than at any other point in history. Having efficient means for accessing this information is essential.

Information retrieval systems are used to search through libraries of books or papers, web pages, and even the entire publicly accessible internet through search engines like Google. One relevant application of this technology is the construction of a search algorithm for the University of Colorado-Boulder website.

The website is a vast corpus of text written by a large number of people. It has a large number of links, a vast and unspecific target audience, and is organized in a nonstandard way, making it very difficult to find important and relevant information. Building an effective indexing and search algorithm will make it easier for students and faculty to find important information quickly. Countless deadlines have been missed, opportunities neglected, and bills left unpaid simply because the relevant web page was tucked away in an obscure corner of the system. A superior search algorithm will make this less likely.

To address this, we will implement a vector space model to index and search the University of Colorado-Boulder's website, using [1]. We have two goals. First, we want to produce relevant results for our queries, which is the search pattern for which we return documents. Second, we wish to reduce the computational intensity in generating our model and performing queries since we must run this model on personal laptops. Special care will be taken to reduce the amount of matrix multiplication needed (or at the least the size of the matrices) as that is known to be quite expensive.

As an overview, we will first explain the mathematical background behind the vector space model for information retrieval (IR). Then, we will explain the mathematical basis upon which

we will reduce the computational requirements for our system. After this, we will demonstrate the IR process on a toy data set, walking through each step in the process from data input to result. Then, we will show three test queries on the corpus gathered from the University of Colorado-Boulder's website and qualitatively examine the results. Finally, we will discuss the efficacy of our model and suggest paths for further research and optimization.

4 Mathematical Formulation

4.1 The Vector Space Model

In the vector space model, each document in a corpus of text is assigned a vector of some dimension based on the contents of the document. The vectors are generated by counting the frequency of each word in the document, assigning the components accordingly, and then normalizing the components, as shown in Table 1. This representation creates a vector space in which, because each dimension corresponds to a particular term, each region of the space corresponds to a particular semantic meaning. Each document's vector extends in the direction of its semantic meaning. Thus, the closer together the two vectors are, the closer their respective meanings. Documents that use similar words will have "similar" semantic meaning vectors and will be closer together in space, as shown in Figure 1. The space will have as many dimensions as there are terms.

Document	**Text in Document**	**Vector Representation**	**Normalized Representation**
0	Cars, Fun	$(1, 1, 0)^T$	$(.7071, .7071, 0)^T$
1	Cars, Monkey	$(1, 0, 1)^T$	$(.7071, 0, .7071)^T$
2	Monkey	$(0, 0, 1)^T$	$(0, 0, 1)^T$
3	Cars, Cars	$(2, 0, 0)^T$	$(1, 0, 0)^T$
4	Monkey, Fun, Monkey	$(0, 1, 2)^T$	$(0, .4472, .8944)^T$

Table 1: Non-normalized Vector Representations for a Corpus of Dimension n

Using Table 1 as an example, The first dimension in the vector representation here corresponds to the term "cars", the second to "fun" and the third to "monkey". The normalized vectors' components are then scaled such that the vector has a length of one under the Euclidean norm. We do this to make future computation easier as it doesn't change the semantic meaning of the vector, since its the geometric relationship between the vectors rather than the vectors themselves that contains the information we need.

Storing the information this way allow us to quickly and easily determine the semantic similarity between documents in a quantitative way. It also allows us to determine those documents which are closest in semantic meaning to a particular query, which is itself assigned a semantic meaning vector, thus creating a word-frequency-based search engine scalable to an arbitrarily large corpus of arbitrary term complexity.

4.2 Querying

A query to the database is a string of terms. This string, just like the documents, can be represented as a vector of n dimensions. For example, using the vector space from Table 1, a

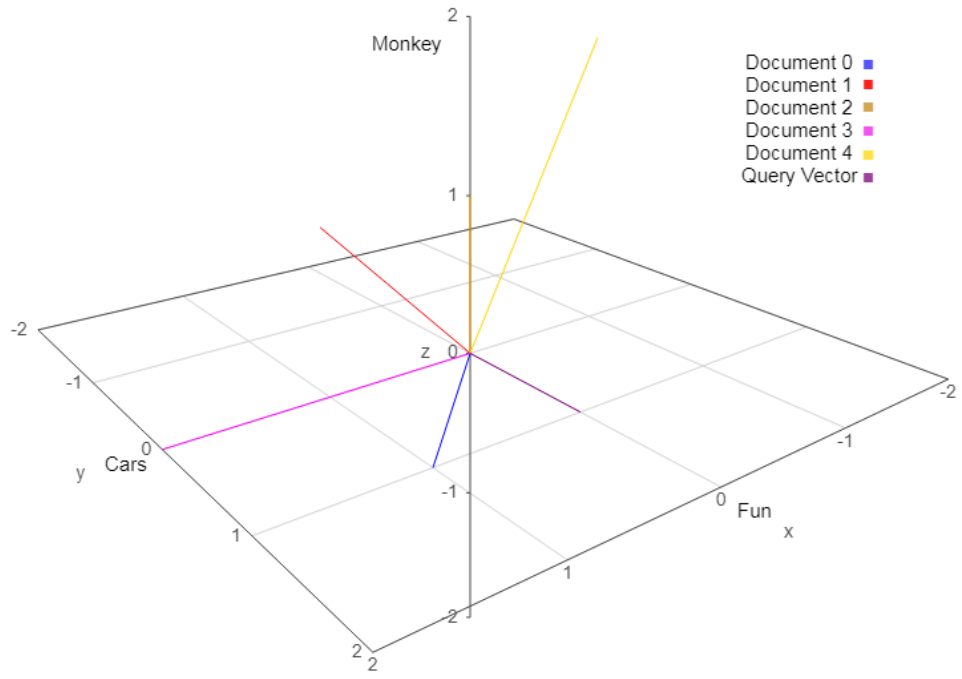

Figure 1: A Visual Representation of Table 1's Document Vectors

query for "fun" would have the vector representation

$$q_1 = (0, 1, 0)^T.$$

A query vector will point in the direction of its semantic meaning, and the document vectors with the least distance to the normalized query vector will correspond to the most topically relevant documents in the corpus.

The geometry of the space can then be exploited to efficiently find which vectors are closest. The cosine of the angle between two unit vectors in the vector space corresponds to how close they are in the space. The closer the cosine is to 1, the closer they are. For a query vector q and any document vector a_j, the cosine of the angle can be calculated using

$$\cos\theta_j = \frac{a_j^T q}{\|a_j\|_2 \|q\|_2}, \tag{1}$$

where $\|x\|_2$ is the Euclidean norm. Thus, to execute a query, simply calculate the cosine between the query's vector and each document's vector representation using Equation 1, and the results with the highest cosine values will be the most relevant.

4.3 Singular Value Decomposition

We will now describe how to use Singular Value Decomposition (SVD) in the context of a vector space model IR. In this section, we will introduce SVD and its key concepts. In future sections, we will show how it is used for IR.

SVD is a factorization of a matrix that breaks up a matrix into three separate matrices such that $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are square matrices and $\mathbf{\Sigma}$ is a diagonal matrix. A pictorial representation of this can be seen in Figure 2

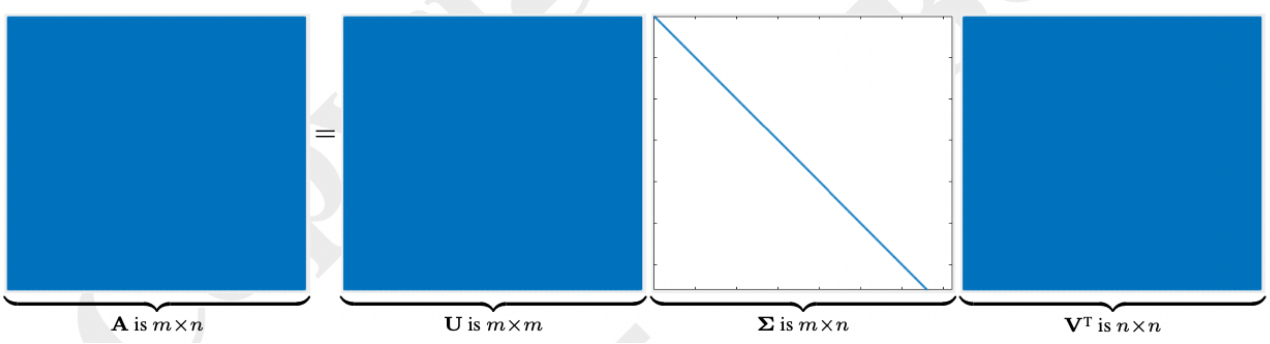

Figure 2: SVD Representation of $\mathbf{A}$ (image from [3])

$\mathbf{\Sigma}$ contains the singular values, which are the square roots of the eigenvalues, of $\mathbf{A}^T\mathbf{A}$. These singular values will be represented in this paper as σ, and they are ordered on the diagonal in increasing order such that $\sigma_{11} \geq \sigma_{22} \geq \cdots \geq \sigma_{\min(m,n)}$. In addition, the columns of $\mathbf{V}$ represent eigenvectors of $\mathbf{A}^T\mathbf{A}$. Similarly, the columns of $\mathbf{U}$ contain the eigenvectors of $\mathbf{A}\mathbf{A}^T$.

Physically interpreted, the columns of $\mathbf{V}$ represent a basis of $\mathbb{R}^n$ and the columns of $\mathbf{U}$ represent a basis of $\mathbb{R}^m$. This means that $\mathbf{U}$ and $\mathbf{V}^T$ are orthogonal matrices. Furthermore, $\mathbf{A}$ represents a way to map basis $\mathbf{V}$ to basis $\mathbf{U}$, meaning:

$$\begin{aligned}\mathbf{A}\,\vec{\mathbf{x}} &= \mathbf{A}(c_1\vec{\mathbf{V}}_1 + c_2\vec{\mathbf{V}}_2 + \cdots + c_n\vec{\mathbf{V}}_n) \\ &= c_1\sigma_{11}\vec{\mathbf{U}}_1 + c_2\sigma_{22}\vec{\mathbf{U}}_2 + \cdots + c_n\sigma_{mm}\vec{\mathbf{U}}_m\end{aligned}$$

4.4 Low-Rank Approximation

We must now discuss two practical points in using the vector space model: dealing with uncertainties as well as methods to make querying computationally cheaper.

As one can imagine, there is no "perfect" way to create a matrix database. There will be variance in how the dictionary of terms is created, how term frequency is calculated, and more. Thus, we could theoretically define an uncertainty matrix $\mathbf{E}$, which could correct for differences in opinion or incomplete information such that the database matrix is more accurately represented by $\mathbf{A}' = \mathbf{A} + \mathbf{E}$. Extrapolating from this, we may then conclude that $\mathbf{A}$ is only closely representative of the "perfect" database matrix, and thus there would be a number of matrices that would represent the data just as well.

Let us assume that $\mathbf{A}$ has rank r_A and that $\mathbf{A}'$ has rank $r_{A'}$ such that $r_{A'} < r_A$. Given the assumption that $\mathbf{A}'$ is a better representation of the true data, this allows us to conclude the data has a matrix approximation of rank $r_{A'}$. Thus, it seems as though rank reduction of the matrix database may help remove noise from the representation. At the very least, it would allow for reasonably accurate querying using far less compute power. To implement this in practice such that the matrix approximation does not lose too much data, we must now establish a few concepts.

4.4.1 Creating a Rank k Approximation with an SVD

First, we need a way to create a low rank-k approximation of $\mathbf{A}$. To do so we will use the concept of SVD, and define a rank-approximation matrix $\mathbf{A}_\mathrm{k} = \mathbf{U}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}\,\mathbf{V}_\mathrm{k}{}^T$. Here, $\mathbf{U}_\mathrm{k}$ is a

$m \times k$ matrix of the first k columns of $\mathbf{U}$, $\mathbf{V}_\mathrm{k}$ is a $k \times n$ matrix of the first k columns of $\mathbf{V}$, and $\mathbf{\Sigma}$ is a $k \times k$ diagonal matrix of with the k largest singular values of $\mathbf{A}$. This is pictorially represented in Figure 3.

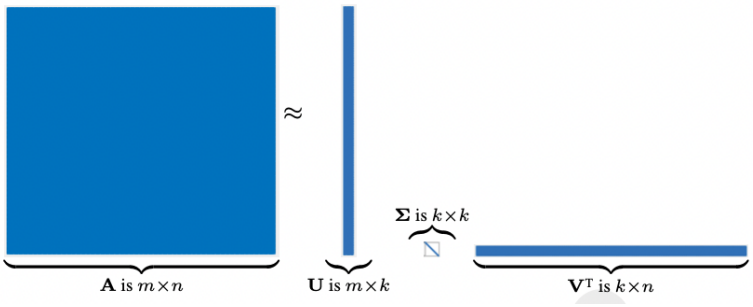

Figure 3: Rank-k Representation of $\mathbf{A}$ (image from [3])

Note that, as stated before, through this approximation, we can represent $\mathbf{A}$ using fewer values, which can conserve computational resources. This works because the singular values (and in turn their corresponding eigenvectors) are in decreasing order, so we can assume that the large elements of $\mathbf{\Sigma}$ and the corresponding columns in $\mathbf{U}$ and $\mathbf{V}$ matter the most in representing $\mathbf{A}$. Mathematically:

$$\begin{aligned} \mathbf{A}\,\vec{\mathbf{x}} &= c_1\sigma_{11}\vec{\mathbf{U}}_1 + c_2\sigma_{22}\vec{\mathbf{U}}_2 + \cdots + c_n\sigma_{mm}\vec{\mathbf{U}}_m \\ &\approx c_1\sigma_{11}\vec{\mathbf{U}}_1 + c_2\sigma_{22}\vec{\mathbf{U}}_2 + \cdots + c_n\sigma_{kk}\vec{\mathbf{U}}_k \qquad \text{(where } k < m\text{)} \end{aligned}$$

4.4.2 A Method to Compare Matrices

Next, to determine how much information is lost in the approximation, we must be able to compare the relative sizes of the matrices. To do this, we use the generalization of the Euclidean norm, the Frobenius matrix norm.

Just as we might divide the Euclidean norms of two vectors to determine the ratio of their lengths, we can do the same with the Frobenius norm. Thus, given some matrices $\mathbf{Y}$ and $\mathbf{X}$, $\|\mathbf{X}\|_F / \|\mathbf{Y}\|_F$ = the ratio of the size between the two. The smaller the ratio, the "smaller" $\mathbf{X}$ is as compared to $\mathbf{Y}$.

With this in mind, let us now derive a computationally efficient way to find the Frobenius norm of an SVD matrix. First, let us assume we have a real $t \times d$ matrix $\mathbf{Y}$. We can define the Frobenius norm in terms of the matrix trace Trace($\mathbf{Y}$), which is the sum of the diagonals of the matrix $\mathbf{Y}^T\mathbf{Y}$, since

$$\begin{aligned} \|\mathbf{Y}\|_F &= \sqrt{\sum_{i=1}^{t}\sum_{j=1}^{d} y_{ij}y_{ij}} \\ &= \sqrt{\mathrm{Trace}(\mathbf{Y}^T\mathbf{Y})} \\ &= \sqrt{\mathrm{Trace}(\mathbf{Y}\mathbf{Y}^T)} \end{aligned} \tag{2}$$

Now we must prove that if the matrix multiplication is defined, multiplying any matrix by an orthogonal matrix does not change the Frobenius norm. Using our $\mathbf{Y}$ and a $t \times t$ orthogonal

matrix $\mathbf{O}$, it follows from Equation 2 that:

$$\begin{aligned} \|\mathbf{OY}\|_F &= \sqrt{\text{Trace}((\mathbf{OY})^T(\mathbf{OY}))} \\ &= \sqrt{\text{Trace}(\mathbf{Y}^T\mathbf{O}^T\mathbf{OY})} \\ &= \sqrt{\text{Trace}(\mathbf{Y}^T\mathbf{Y})} \\ &= \|\mathbf{Y}\|_F \end{aligned} \tag{3}$$

It follows from a similar argument that given a $d \times d$ orthogonal matrix $\mathbf{V}$:

$$\|\mathbf{YV}\|_F = \|\mathbf{Y}\|_F \tag{4}$$

Using these results, since $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, it follows from Equation 3 and Equation 4, that the Frobenius norm of a SVD is:

$$\begin{aligned} \| \mathbf{A} \|_F &= \| \mathbf{U} \, \mathbf{\Sigma} \, \mathbf{V}^T \|_F \\ &= \| \mathbf{\Sigma} \, \mathbf{V}^T \|_F \\ &= \| \mathbf{\Sigma} \|_F \\ &= \sqrt{\sum_{i=1}^{r_A} \sigma_{ii}^2} \end{aligned} \tag{5}$$

Thus, once we compute the SVD of a matrix, we simply need to sum the squares of the singular values, which makes computing the Frobenius norm of our matrix much faster.

4.4.3 Determining a Valid Rank for Rank Approximation

Now that we have a mechanism to compare size, we need one final piece, which will then allow us to create a heuristic to choose which rank-k approximation we ought to use to represent our data set.

For this computation, we will need to be able to find $\| \mathbf{A} - \mathbf{A}_\text{k} \|_F$ as this gives the "size" of the information loss/change in using the rank-k approximation, which we can then compare against the full matrix. This derivation is outside the scope of this project, and thus we will simply state that according to the Eckart–Young–Minsky Theorem [2]:

$$\| \mathbf{A} - \mathbf{A}_\text{k} \|_F = \sqrt{\sum_{d=k+1}^{r_A} \sigma_{dd}^2} \tag{6}$$

Hence, rather than take the difference and then compute the entire Frobenius norm, we need only use the singular values left over from the partition.

Now we can put this all together to determine a computationally efficient way to find an acceptable approximation of our database. We use the following algorithm:

1. *denominator* := $\| \mathbf{A} \|_F$ using Equation 5
2. For $x := r_A$ until $x = 1$:
   (a) *numerator* := $\| \mathbf{A} - \mathbf{A}_\text{k} \|_F$ using Equation 6

(b) $result := \frac{numerator}{denominator}$

(c) If *result* is too large (in our case ≥ 0.4), halt the algorithm and use $k = x - 1$ as the rank for our approximation

(d) $x := x - 1$

As seen above, we first store the Frobenious norm of the matrix, so we only have to compute it once. Then, for each value of k, we compute the Frobenius norm of the difference between our approximation and the actual matrix. Then we compute the ratio of the difference to the actual matrix, and this then tells us how small our change in the data size would be if we were to use the approximation in place of the true matrix. The larger the size, the more data we are potentially losing. Thus, when that change becomes too large (for some agreed bound), we know what approximation we can use. We can now define our querying method in terms of a rank-k approximation of the database matrix.

Note that the cutoff value is generally matrix/implementation specific and can be difficult to decide upon. For the purposes of this paper, we will assign the limit of 0.4 as we are willing to sacrifice some accuracy in pursuit of computational ease. Know however, that is is more usual to use a value of 0.2 or smaller [1].

4.5 Using The SVD Rank k Approximation for Querying

Finally, we can determine how to resolve a query using this approximation. Let us define e_j as the jth column of the $n \times n$ identity matrix, so that the jth column of $\mathbf{A}_\mathrm{k}$ is equivalent to $\mathbf{A}_\mathrm{k}\, e_j$. Building upon the querying method described in Section 4.2, we thus could say that the cosine of the angle between a query q and the jth approximated document vector is

$$\begin{aligned} \cos\theta_j &= \frac{(\mathbf{A}_\mathrm{k}\, e_j)^T q}{\|\,\mathbf{A}_\mathrm{k}\, e_j\|_2 \|q\|_2} \\ &= \frac{(\mathbf{U}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}\,{\mathbf{V}_\mathrm{k}}^T e_j)^T q}{\|\,\mathbf{U}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}\,{\mathbf{V}_\mathrm{k}}^T e_j\|_2 \|q\|_2} \\ &= \frac{e_j^T\,\mathbf{V}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}({\mathbf{U}_\mathrm{k}}^T q)}{\|\,\mathbf{U}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}\,\mathbf{V}_\mathrm{k}\, e_j\|_2 \|q\|_2} \end{aligned} \tag{7}$$

This is the equation with which we will compute our query results.

5 Examples and Numerical Results

5.1 Matrix and Dictionary Generation

To generate the term-document matrix for a set of documents, we begin by creating two lists in python corresponding to our term list, called `words`, and our list of document URLs, called `documents`. Each index in `words` will correspond to a row in the term-document matrix. Initially, this string will be empty. Each index in `documents` will correspond to a column in the term-document matrix where the document's vector resides.

To generate a vector for each document, the document is scanned, and the frequency of every word contained within it is recorded in a Python dictionary. During this process, all

words are stemmed and made lower-case, meaning they are converted to their most generic form. For example, the words "bake" and "baking" would both be converted to the word "bake". For this, we use the `nltk.stem` library in Python. Here, we also filter out any articles like "a" and "the", punctuation, special characters, and contractions, replacing the latter with their expanded form.

Next, a list called `column` is created to hold the vector's components. `words` is looped through, and the dictionary is checked for each term to see if it is in the dictionary. If the term is present, the frequency of the term is appended to `column`, and that term is removed from the document's dictionary. If it is not, a zero is appended (meaning that a particular word is not present in the dictionary). Finally, we check if the dictionary is empty. If it is, all words in the document have already been encountered, and no new dimensions are necessary. If it is not, then new words have been encountered. These new words are appended to `words`, and their frequencies appended to `column` as new components

This is done for each document. At the end, we have a set of document vector component lists of varying sizes. This is because, for each document with new words, additional components are added to its vector that were not needed for previous vectors. As this is not useful for making a matrix, we append zeros to all document vectors except the last one calculated until they all have the same number of components as that last vector. Thus, all document vectors will have the same dimension. Finally, a matrix $\mathbf{A}$ is constructed, with the document vector component lists being the columns The code for this entire process can be found in (Section 7.2.3).

As an example, we generated a corpus of five documents with six terms shown in Table 2. Here, the six dimensions of the vector space correspond to the terms 'cars', 'fast', 'fun', 'crazy', 'monkey', and 'swing', in the order in which they appeared in the corpus.

Document	**Text in Document**	**Vector Representation**	**Normalized Representation**
0	Cars, Fast, Fun	$(1,1,1,0,0,0)^T$	$(.5774,.5774,.5774,0,0,0)^T$
1	Cars, Crazy, Monkey	$(1,0,0,1,1,0)^T$	$(.5774,0,0,.5774,.5774,0)^T$
2	Crazy, Monkey	$(0,0,0,1,1,0)^T$	$(0,0,0,.7071,.7071,0)^T$
3	Cars, Cars, Fast, Fun	$(2,1,1,0,0,0)^T$	$(.8165,.4082,.4082,0,0,0)^T$
4	Monkey, Swing, Fun	$(0,0,1,0,1,1)^T$	$(0,0,.5774,0,.5774.5774)^T$

Table 2: Toy Data Set for Explanation

Following the process above generates the matrix $\mathbf{A}$ as shown in Equation 8.

$$\mathbf{A} = \begin{bmatrix} .5774 & .5774 & 0 & .8165 & 0 \\ .5774 & 0 & 0 & .4082 & 0 \\ .5774 & 0 & 0 & .4082 & .5774 \\ 0 & .5774 & .7071 & 0 & 0 \\ 0 & .5774 & .7071 & 0 & .5774 \\ 0 & 0 & 0 & 0 & .5774 \end{bmatrix} \tag{8}$$

5.2 Rank Reduction

Unlike this test matrix, the standard database would be much larger. For example, We performed this process on a set of 4381 pages from the CU website, which is described in Section 5.4, and the matrix had a dimension of 24799×4381. In these cases, it is necessary to

reduce the complexity of querying to make doing so feasible. We do this by applying Singular Value Decomposition to the matrix as explained in Section 4.3 and, and then using the algorithm in Section 4.4.3 to find the appropriate rank to reduce to. Then we apply the low-rank approximation.

This is unnecessary in the case of of example data $\mathbf{A}$, but we will reduce the matrix to rank 3 as a demonstration. We first calculate the SVD in the form $\mathbf{A} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T$ using the code in Section 7.2.5:

$$\mathbf{U} = \begin{bmatrix} -.6049 & -.3761 & -.4191 & -.5630 & -.8468 & 0 \\ -.2967 & -.3799 & -.0677 & .6230 & -.5978 & -.1328 \\ -.4309 & -.2990 & .5256 & .2715 & .5978 & .1328 \\ -.3412 & .5141 & -.3851 & .3098 & .0740 & .6079 \\ -.1341 & .0808 & .5932 & -.3515 & -.5238 & .4750 \end{bmatrix}, \tag{9}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 1.6077 & 0 & 0 & 0 & 0 \\ 0 & 1.2465 & 0 & 0 & 0 \\ 0 & 0 & .8635 & 0 & 0 \\ 0 & 0 & 0 & .3397 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{10}$$

$$\mathbf{V}^T = \begin{bmatrix} -.4785 & -.5104 & -.3591 & -.4919 & -.3736 \\ -.4887 & .3395 & .6291 & -.4687 & .1745 \\ .0259 & -.3985 & -.1449 & -.1798 & .8873 \\ .5633 & -.5011 & .5580 & -.2782 & -.2068 \\ -.4629 & -.4629 & .3780 & .6547 & 0 \end{bmatrix}. \tag{11}$$

We then calculate the low-rank approximation $\mathbf{A}_\mathrm{k}$. In this example, we will use $k = 3$ to generate the approximation in the form $\mathbf{A}_\mathrm{k} = \mathbf{U}_\mathrm{k}\,\mathbf{\Sigma}_\mathrm{k}\,{\mathbf{V}_\mathrm{k}}^T$ where $\mathbf{U}_\mathrm{k}$ contains the first k columns of $\mathbf{U}$, $\mathbf{V}_\mathrm{k}$ contains the first k columns of $\mathbf{V}$, and $\mathbf{\Sigma}$ is a $k \times k$ diagonal matrix with the k largest singular values of $\mathbf{A}$ (see Section 7.2.5 for the code which does this).

$$\mathbf{U}_\mathrm{k} = \begin{bmatrix} -.6049 & -.3761 & -.4191 \\ -.2967 & -.3799 & -.6766 \\ -.4309 & -.2990 & .5256 \\ -.3412 & .5141 & -.3851 \\ -.4754 & .5949 & .2081 \\ -.1341 & .8081 & .5932 \end{bmatrix} \tag{12}$$

$$\mathbf{\Sigma}_\mathrm{k} = \begin{bmatrix} 1.6077 & 0 & 0 \\ 0 & 1.2465 & 0 \\ 0 & 0 & .8635 \end{bmatrix} \tag{13}$$

$${\mathbf{V}_\mathrm{k}}^T = \begin{bmatrix} -.4785 & -.5104 & -.3591 & -.4919 & -.3736 \\ -.4887 & .3395 & .6291 & -.4687 & .1745 \\ .0259 & -.3985 & -.1449 & -.1798 & .8873 \end{bmatrix} \tag{14}$$

This yields the rank-reduced matrix $\mathbf{A}_\mathrm{k}$.

$$\mathbf{A}_{\mathrm{k}} = \begin{bmatrix} .6851 & .4815 & .1067 & .7633 & -.0396 \\ .4581 & .1061 & -.1181 & .4671 & .0438 \\ .5254 & .0462 & -.0515 & .4339 & .5964 \\ -.0593 & .6301 & .6484 & .0293 & .0218 \\ .0080 & .5703 & .7150 & -.0039 & .5744 \\ .0673 & -.0598 & .0666 & -.0332 & .5527 \end{bmatrix} \tag{15}$$

5.3 Querying the Example Data

We now have the final form of the matrix corresponding to our corpus. To perform a search, we merely take all the terms in a search string, say "monkey", and create a query vector (see Section 7.2.6 for the code). We iterate through the query, and for every word which exists in the query string, we place a 1 in the corresponding query vector component, placing zeros everywhere else. For example, the query "monkey" will have the vector representation

$$q = (0, 0, 0, 0, 1, 0)^T.$$

We then compute the cosine of the angle between the query vector and each one of the document vectors using Equation 7, and sort them from least to greatest. A cosine of 1 corresponds to a perfect match (this would mean a parallel vector). For example, When the "fun" query is executed, the results are shown in Table 3

Document	**Text in Document**	**Cosine**
2	Crazy, Monkey	0.7282
1	Cars, Crazy, Monkey	0.5787
4	Monkey, Swing, Fun	0.5758
0	Cars, Fast, Fun	0.0081
3	Cars, Cars, Fast, Fun	-0.0040

Table 3: The Results of the Query "fun" on the Example Data Set

We can see that the three first results with the highest cosine value are the three results containing the search term. These all have cosine values above .5, with the first result having one above .7 as "monkey" is one of only two keywords instead of three, making it more important. Naturally, the two documents without the search term have cosine values near zero. This is a good proof of concept for our query model.

5.4 The CU Website

Unlike the example data set, we did not hand make the CU data used. Instead, we had to get that data from the internet. To do this, we first used `sitemap-generator` to generate a sitemap of 4,381 links to pages on the CU website by crawling four layers deep from the homepage `www.colorado.edu` (see Section 7.2.1). We then used `scrapy` to get the text from those links and saved them in a machine-readable format called JSON, with each link paired with its text (see Section 7.2.2). Then, using the same techniques outlined above, we calculated a normal vector for each of the documents in the corpus and formed a matrix of dimension

24799×4381, with $24,799$ corresponding to the number of unique terms encountered in the corpus.

We then performed SVD on this matrix and used the SVD rank-k approximation technique discussed previously, using rank 400. To determine the rank value of 400, we ran the algorithm outlined in Section 4.4.3 until we hit our bound of 0.4. Again, this is a rather high bound, but it saved us a lot of computation, so we accepted the trade-off.

We ran three queries: one for "music innovation", one for "entrepreneurship venture capital", and one for "sports medicine enrollment". The top 10 results are shown in Table 4, Table 5, and Table 6, respectively. The corresponding URLs to these page titles can be found in Section 7.1.

Cosine	**Page Title**
.5072	Facilities - College of Music
.5044	About Us - College of Music
.5032	Giving News - College of Music
.4915	Summer Session - College of Music
.4908	Bachelor of Arts in Music
.4749	Expanded Imig Building - College of Music
.4729	Composition news - Page 6 - College of Music
.4719	Noteworthy - College of Music
.4643	College of Music
.4640	Minor in Music

Table 4: The Results of the Query "music innovation" on the CU Web Data

When searching for "music innovation", we can see the most common results are those from the College of Music. This is likely because the vast majority of mentions of the term "music" come from the College of Music. The term "innovation", however, is more general, so can appear on the website for any college or any subject that is engaged in "innovation". Thus, the documents whose vectors are most closely aligned with the query vector must all be from the College of Music. These results make sense and demonstrate the efficacy of our model.

When searching for "entrepreneurship venture capital", the results are very topical. The top results all come from pages that explicitly have to do with innovation, entrepreneurship, and venture capital–the first, third, eighth, and ninth–or have to do with both innovation and entrepreneurship. Lots of results are also from the Innovation & Entrepreneurship section of the CU website, which is exactly what you would want in a search engine. Although the cosine scores are lower than for "music innovation", the results are still relevant, meaning the high cosine scores in that query likely came from the repetition of the word "music".

Cosine	Page Title
.2653	New Venture Challenge powers campus entrepreneurship
.2608	About - Innovation & Entrepreneurship
.2593	Groundbreaking CU Boulder innovators awarded $1.5 million in grants at Lab Venture Challenge finals - Venture Partners at CU Boulder
.2585	Innovation & Entrepreneurship
.2379	Entrepreneurial Programs - Innovation & Entrepreneurship
.2241	Faculty & Research Staff - Innovation & Entrepreneurship
.2170	Upcoming Entrepreneurial Events - Innovation & Entrepreneurship
.2155	Funding awarded to top student female founders at women's prize night - New Venture Challenge
.2142	CU Boulder inventions transformed to meet market needs after commercialization program - Venture Partners at CU Boulder
.2141	Innovation & Entrepreneurship Faculty & Staff Resources - Innovation & Entrepreneurship

Table 5: The Results of the Query "entrepreneurship venture capital" on the CU Web Data

Cosine	Page Title
.2998	Certificate in Critical Sport Studies
.1965	Minor in Sports Media
.1870	EXPERTS: Tokyo Olympic Games - CU Boulder Today
.1795	Recreation Services
.1663	Winter Olympics and CU Boulder Olympians - CU Boulder Today
.1587	Bachelor of Arts in Integrative Physiology
.1472	With 20k award, undergrad advances inclusion in sports - Colorado Arts and Sciences Magazine
.1424	Equipment - Recreation Services
.1304	Register for Classes - Office of the Registrar
.1260	Athletics - Page 9 - CU Boulder Today

Table 6: The Results of the Query "sports medicine enrollment" on the CU Web Data

Finally, when searching for "sports medicine enrollment", as there is no specific "sports medicine" program on campus, the cosine scores are the lowest for this query. However, the results get close. Three results have to do with sports-related programs, two results have to do with athletic competitions, and the rest are related to athletics or enrollment in some other way. Even though this query could not turn up something specifically accurate by design, it got extremely close. This proves the efficacy of our model even in non-ideal situations.

6 Discussion and Conclusions

The vector space model we constructed for information retrieval on the CU website worked quite well! For each of the three queries we ran, the search results that came up were relevant and useful, even in a situation where no perfectly relevant results existed. The SVD rank

reduction process did not substantially reduce the algorithm's performance while substantially decreasing execution time. These are the results we expected.

This demonstrates the effectiveness of vector space models for indexing and querying a large corpus of text and of SVD for reducing the term-document matrix's rank and thus search complexity. If desired, this means that our model could be used in new search functionality for the website.

The execution time for querying documents, however, is quite long. At the moment, we are still creating the full rank SVD representation of the matrix when we really only need certain values from the decomposition. In the future, we hope to explore more efficient computation methods so that we do not need to compute the full SVD and can rather selectively compute the values we need.

In addition, we would like to compare our IR system to other options quantitatively rather than qualitatively. This could be done by indexing a competitive querying data set as shown in [4] and testing the accuracy and speed of our model against other top performing methods.

On top of that, in other vector space models implemented in the literature as seen in [1], terms were given *global* weights across the database in addition to the local frequency-based weights used here. These were useful as they could reduce the impact of terms that appear consistently (i.e., "university" for a university website), thus increasing the relative importance of rarer terms. Consequently, the model developed here could be further improved by adding this global weighting functionality based on the prevalence of different words and limiting the number of terms collected from each document (by focusing on titles and metadata only).

Finally, the CU data set could have been better cleaned. The pages we grabbed contained a substantial amount of noise (headers, navigation bar links, nonsense words, etc.) which reduced the efficacy of searches. Although the model still performed quite well under these circumstances, cleaner data would no doubt increase the search abilities of the model and reduce the complexity of the problem at hand.

7 Appendix

7.1 Tables with URLs

Cosine	URL
.5072	https://www.colorado.edu/music/about-us/facilities
.5044	https://www.colorado.edu/music/about-us
.5032	https://www.colorado.edu/music/news/giving-news
.4915	https://www.colorado.edu/music/academics/summer-session
.4908	https://www.colorado.edu/academics/ba-music
.4749	https://www.colorado.edu/music/giving/expanded-imig-music-building
.4729	https://www.colorado.edu/music/news/composition-news?page=5
.4719	https://www.colorado.edu/music/news/noteworthy
.4643	https://www.colorado.edu/music
.4640	https://colorado.edu/academics/minor-music

Table 7: The URLs Which Correspond to the Query: "music innovation"

Cosine	URL
.2653	https://www.colorado.edu/research/report/2016-17/new-venture-challenge-powers-campus-entrepreneurship
.2608	https://www.colorado.edu/innovate/about
.2593	https://www.colorado.edu/venturepartners/2021/11/08/groundbreaking-cu-boulder-innovators-awarded-15-million-grants-lab-venture-challenge
.2585	https://www.colorado.edu/innovate/
.2379	https://www.colorado.edu/innovate/entrepreneurial-programs
.2241	https://www.colorado.edu/innovate/faculty-research-staff
.2170	https://www.colorado.edu/innovate/events
.2155	https://www.colorado.edu/nvc/2022/03/09/funding-awarded-top-student-female-founders-womens-prize-night
.2142	https://www.colorado.edu/venturepartners/2022/02/03/cu-boulder-inventions-transformed-meet-market-needs-after-commercialization-program
.2141	https://www.colorado.edu/innovate/faculty-staff-resources

Table 8: The URls Which Correspond to the Query: "entrepreneurship venture capital"

Cosine	URL
.2998	https://colorado.edu/academics/certificate-critical-sport-studies
.1965	https://www.colorado.edu/academics/minor-sports-media
.1870	https://www.colorado.edu/today/experts-tokyo-olympic-games
.1795	https://www.colorado.edu/recreation/
.1663	https://www.colorado.edu/today/winter-olympics-and-cu-boulder-olympians
.1587	https://www.colorado.edu/academics/ba-integrative-physiology
.1472	https://www.colorado.edu/asmagazine/2021/12/01/20k-award-undergrad-advances-inclusion-sports
.1424	https://colorado.edu/recreation/facilities/student-recreation-center/equipment
.1304	https://www.colorado.edu/registrar/students/registration/register
.1260	https://www.colorado.edu/today/athletics?page=8

Table 9: The URLs Which Correspond to the Query: "sports medicine enrollment"

7.2 Code for Reference

7.2.1 Sitemap Generator

```
const SitemapGenerator = require('sitemap-generator');

//NOTES:
// https://docs.npmjs.com/downloading-and-installing-node-js-and-npm
//h ttps://www.npmjs.com/package/sitemap-generator

//TO USE:
// run using node sitemapGenerator.js


// create generator
const generator = SitemapGenerator('https://www.colorado.edu/', {
  maxDepth: 4,
  maxEntriesPerFile: 10000,
  stripQuerystring: false
});

var i = 0;

// register event listeners
generator.on('done', () => {
  console.log("Done.");
  // say done upon completion
});
```

```
// every 50 adds, print how many resources have been added
generator.on('add', (url) => {
  i += 1;
  if (i % 50 == 0) {
    console.log("Added " + i + " resources");
  }
});

// output all errors
generator.on('error', (error) => {
  console.log(error);
});

// start the sitemap generator
generator.start();

// saves output to sitemap.xml
```

7.2.2 Scraper

NOTE: this code is what one slots into the scrapy boilerplate execution environment, but is not all of the program. See https://scrapy.org for more details.

```
import scrapy
import xml.etree.ElementTree as ET
from os.path import exists
import json

# function to open json file and add document to text-data.json file using correct syntax.
def write_json(data, filename="text-data.json"):
    with open(filename,'r+') as file:
        file_data = json.load(file)
        file_data["data"].append(data)
        file.seek(0)
        json.dump(file_data, file, indent = 4)

# to run spider, navigate to directory above this one and run "scrapy crawl search".
class SearchSpider(scrapy.Spider):
    name = "search"

    def start_requests(self):
        # create text-data.json if it doesn't exist.
        if not exists("text-data.json"):
            with open("text-data.json", 'w') as file:
                file.write('{"data":[]}') # prime for data input.
```

```
    start_urls = [] # list of all encountered urls in order.

    # parse the sitemap XML file generated previously.
    print("Begin Parsing:")
    mytree = ET.parse('../sitemap.xml')
    myroot = mytree.getroot()
    for child in mytree.iter():
        #print(child.tag)
        if child.tag == "{http://www.sitemaps.org/schemas/sitemap/0.9}loc":
            print(child.text)
            start_urls.append(child.text)
    print("Parsing complete.")

    # start_urls now contains all urls for the corpus.

    i = 0
    for url in start_urls: # for every url, scrape usng the parse funciton.
        i += 1
        yield scrapy.Request(url=url, callback=self.parse, meta={'url': url, 'i': i})

# parse a url function
def parse(self, response):
    # print to console the document number
    self.log("I : " + str(response.meta.get('i')))
    ignore_tags = ['.menu-icon-text'] # classes and tags to ignore
    req_tags = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'p', 'span', 'strong']
    # ^ all HTML tags to capture text from

    section_selector = response.css('body') # scrape from HTML tag body
    for section in section_selector: # iterate through all sections inside body
        texts = [] # list to hold texts
        for tag in section.css('*'): # for every tag in body
            if tag.root.tag in ignore_tags: # if the tag is in ignore tags, skip
                tag.root.getparent().remove(tag.root) # remove all children from the tree
        for tag in section.css('*'): # for every tag in body
            if tag.root.tag in req_tags: # if the tag is one of the tags to capture from
                texts = texts + tag.css('*::text').getall() # add all text to texts
        write_json({"url": response.meta.get('url'), "texts": texts}) # write the doc and
            ↪ al
                                                                # saved data to the json

        self.log("logged")
```

7.2.3 Dictionary and Initial Matrix Generator

```
from __future__ import print_function
import json
import numpy as np

#nltk imports
import nltk
nltk.data.path.append("/home/gigabyte/Desktop/matrixMethods/project/Info-Retireval-3310/
    ↪ nltk")
nltk.data.path.append("/home/becketth/3310/Info-Retireval-3310/nltk")
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
import nltk.stem as stem

from collections import Counter

# filename gets adjusted depending on where the dataset is saved.
filename = "search_spider/demo-data.json"

# the set of words to ignore
stopWords = set(stopwords.words('english'))

# the library to reduce words to their most basic form
stemmer = stem.PorterStemmer()

documents = [] # holds list of document urls, index corresponds to doc's col
words = [] # holds list of words, index corresponds to word's row

matrixArray = []

with open(filename, "r") as file:
    file_data = json.load(file) # extracts JSON data from the file
    for document in file_data["data"]:
        documents.append(document["url"])
        # saves page url in index which corresponds to its column

        doc_words = {}

        doc_raw = word_tokenize(" ".join(document["texts"]).lower())
        # compiles the whole document into one lowercase array of terms.
        for w in doc_raw:
            w = stemmer.stem(w) # reduces the term to its simplest form
            if w not in stopWords: # if not an ignored word
                if w not in doc_words: # if the word has not already been encountered
                    if w.isalpha()==True: # ignore punctuation
```

```
                doc_words[w] = 1 # add the term w/ a count of 1 to this doc's dictionary
                    ↪ .
            else: # if it has been encountered
                doc_words[w] += 1 # increment the count by 1

        column = [] # list for the vector components to be inserted into

        for i in range(len(words)): # iterates over all found words
            if words[i] in doc_words: # if the word is in the doc
                column.append(doc_words[words[i]]) # add its frequency as that word's
                    ↪ component
                doc_words.pop(words[i]) # remove the word from the doc's dictionary
            else: # if the word isn't in the doc
                column.append(0) # set that word's component to zero.
            i += 1

        for uncaughtkey in doc_words: # for all novel words in the doc
            words.append(uncaughtkey) # add the word to the list of all words
            column.append(doc_words[uncaughtkey]) # add the frequency as that word's
                ↪ component

        norm = np.linalg.norm(column, ord=2) # get the length of the vector
        normed_col = []
        for c in column: # normalize all vector components
            if norm == 0:
                normed_col.append(0) # in case the document was empty (zero vector).
            else:
                normed_col.append(c/norm)


        matrixArray.append(normed_col) # add the normalized column to the matrix.
# print(matrixArray)

np.savetxt("documents.txt", documents, delimiter =", ", fmt ="% s") # save documents to
    ↪ text file for later
np.savetxt("words.txt", words, delimiter=", ", fmt="% s") # save words to text file for later

for doc in matrixArray: # the matrix right now is jagged. Ensure that each vector has
                        # as many dimensions as there are words, filling missing
                        # dims with zeros.
    d = (len(words) - len(doc))
    if (d > 0):
        for i in range(d):
            doc.append(0)

# print(matrixArray)
```

```
print("++++++++++==")
M = np.transpose(np.array(matrixArray)) # cols are actually rows, fix by transposing.

print(M)

np.savetxt('pre-svd.txt', M) # save matrix for future use.
```

7.2.4 Rank Choice

```
import numpy as np

# load the original, pre-rank-reduced matrix from SVD.
D = np.loadtxt('D-full-big.txt')

# set the maximum possible rank
rank = len(D)
# set the perfect ratio
ratio = 0

# perform the algorithm rom section 4.4.3.
# minimum rank of one and max ratio of .4
while rank > 1 and ratio < .4:
    rank -= 1 # reduce rank by one
    denom = 0
    # sum squares of singular values up to rank
    for i in range(rank):
        denom += (D[i][i])^2
    # take square root
    denom = np.sqrt(denom)

    num = 0
    # sum squares of singular values after rank
    for i in range (rank, len(D)):
        num += (D[i][i])^2
    # take square root
    num = np.sqrt(num)

    # calculate ratio
    ratio = num/denom

# return the correct rank.
print(str(ratio) + " at rank "+ str(rank))
```

7.2.5 Rank k Matrix Creation

```
import numpy as np

A = np.loadtxt('pre-svd.txt', dtype=float) # load the term-document matrix

print("Loaded matrix.")
# print(A)

U, first_D, V_t = np.linalg.svd(A) # perform SVD on the matrix

print("Svd complete.")

# the D matrix is returned as a string
# this converts it to a diagonalized matrix
D = np.zeros((U.shape[1], V_t.shape[0]))
D[:first_D.size, :first_D.size] = np.diag(first_D)

np.savetxt('D-full.txt', D)

print("Reducing rank.")

# the rank to reduce the matrix to.
# 3 for the example dataset
# 400 for the CU website dataset
rank = 3

# perform rank reduction on the matrix components
U = np.delete(U, slice(rank, U.shape[1]), axis=1)
D = np.delete(D, slice(rank, D.shape[1]), axis=1)
D = np.delete(D, slice(rank, D.shape[0]), axis=0)
V_t = np.delete(V_t, slice(rank, V_t.shape[0]), axis=0)

print("Rank appx:")
rankAppx = np.matmul(U, np.matmul(D, V_t)) # finds the A_k matrix.

print(rankAppx)

# saves all useful components for later.
np.savetxt('rank-appx.txt', rankAppx)
np.savetxt('U-appx.txt', U)
np.savetxt('D-appx.txt', D)
np.savetxt('Vt-appx.txt', V_t)
```

7.2.6 Querying

```
import numpy as np

import nltk.stem as stem

# the library to reduce words to their most basic form
stemmer = stem.PorterStemmer()

# loads the document and word arrays
documents = np.loadtxt('documents.txt', dtype=str, delimiter='\n')
words = np.loadtxt('words.txt', dtype=str, delimiter='\n')

# loads the components of the reduced SVD and the final matrix
U = np.loadtxt('U-appx.txt')
D = np.loadtxt('D-appx.txt')
V_t = np.loadtxt('Vt-appx.txt')
rankAppx = np.loadtxt('rank-appx.txt')

# query function
def query(querytext):

    query = [] # array to hold the vector form of the query

    # splits the query string into an array,
    # reduces the words to their most basic form
    # and saves them in the stemmed list.
    s = querytext.split()
    stemmed = []
    for ss in s:
        stemmed.append(stemmer.stem(ss.lower()))

    # iterates through all the words in the corpus. If the word is in the query,
    # that word's dimension is set to 1 in the query vector. If not, zero.
    for i in range(len(words)):
        if words[i] in stemmed:
            query.append(1)
        else:
            query.append(0)

    # converted to a column vector.
    query = np.transpose(query)

    # list to hold the cosine values for each document (in order)
    similarities = []
```

```
    # for every document vector
    for col in range(rankAppx.shape[1]):
        # creates vector corresponding to index−matrix column of index−matrix
        # of size dxd where d is the number of documents
        e = np.zeros(rankAppx.shape[1])
        e[col] = 1
        e = np.transpose([e])

        # gets transpose of V_t
        V = np.transpose(V_t)

        # gets transpose of U
        U_t = np.transpose(U)

        # calculates the numerator and denominator in the cosine equation.
        num = np.matmul(np.matmul(np.matmul(np.transpose(e), V), D), np.matmul(U_t, query))
        denom = np.linalg.norm(np.matmul(np.matmul(D, V_t), e), ord=2) * np.linalg.norm(query
            ↪ , ord=2)

        if denom == 0: # avoid divide by zero
            costheta = [−1] # if divide by zero, assign cosine of least relevance
        else:
            costheta = num/denom

        # add the costheta value at the index corresponding to its document.
        similarities.append(costheta[0])

    return similarities # return results

# run query and save cosine values in order (for example query "monkey")
resp = query("monkey")

# array for tuples of cosine value and document URL/title
results = []
i = 0
for res in resp: # iterate through all the cosine values
    if res > −2: # sanity check
        results.append((res, documents[i]))
    i += 1

# sort the results by cosine value
results = sorted(results, key = lambda x: x[0])

# print the results to the terminal
for result in results:
    print(str(result[0]) + " :: " + str(result[1]))
```

```
# save the results for later
np.savetxt("results.txt", results, delimiter=", ", fmt="% s")
```

References

[1] Michael Berry, Zlatko Drmac, and Elizabeth Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–351, April 1999.

[2] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[3] University of Colorado-Boulder Applied Mathematics Department. Analyzing the Mariana Trench, 2022.

[4] University of Glasgow Information Retrieval Group. CISI. `https://www.kaggle.com/datasets/dmaso01dsta/cisi-a-dataset-for-information-retrieval?select=CISI.ALL`, 2020. Accessed: 2022-06-25.