A STUDY ON MATRIX FACTORIZATION AND ITS APPLICATIONS

TANG WEN KAI, ADRIAN

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Applied Mathematics with Computing

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

September 2021

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : ____________________

Name : Tang Wen Kai, Adrian

ID No. : 17UEB05957

Date : 26/08/2021

APPROVAL FOR SUBMISSION

I certify that this project report entitled "**A STUDY ON MATRIX FACTORIZATION AND ITS APPLICATIONS**" was prepared by **TANG WEN KAI, ADRIAN** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Science (Honours) Applied Mathematics with Computing at Universiti Tunku Abdul Rahman.

Approved by,

Signature : ________________

Supervisor : Ng Wei Shean

Date : 26/8/2021

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2021, TANG WEN KAI, ADRIAN. All rights reserved.

ACKNOWLEDGEMENTS

First, I would like to express my gratitude to my supervisor, Dr Ng Wei Shean for giving me the opportunity to do research related to matrix factorizations and their application in real-world problem. She guides and gives advice that help in completing the research and report. She also introduced Latex to me, so that I could use it for writing mathematical report.

Besides, I also would like to thank Dr Liew How Hui for guiding and advising me regarding Python coding.

Lastly, I would like to thank my family for the financial support. If not I would not be able to complete this final year project.

ABSTRACT

Matrix factorizations are methods used to factorise a matrix into a product of two or more matrices. Each matrix factorizations have their own properties respectively. Matrix factorization is mostly used in image processing and recommendation systems. Both applications use high dimension matrices to calculate the result. This is where matrix factorizations are used to reduce dimension of the data set that help in reducing the computational power. In this project, we focus on Singular Value Decomposition (SVD) and Non-Negative Matrix Factorization (NMF) applied in Latent Semantic Indexing (LSI).

In order to carry out the project, we first read intensively on other research papers to increase the knowledge related to SVD and NMF. We study the computational steps, properties and application in the real-world problems. Computational steps are important as it serves the basic knowledge to code it in Python. Python also consists of libraries that can be used to calculate the approximated matrix with some parameter tuning.

In this project, the application that we focus on is LSI algorithm. LSI is a search algorithm where it returns a set of documents that is related to the keywords that the user searches. It required high computational power to do matrix multiplication. To solve this, we used SVD and NMF methods to reduce the matrix dimension and thus reduce the computational power. SVD performed better than NMF because SVD has the appropriate method to find the dimension to reduce whereas NMF does not have that kind of method. In the future, we can find methods that can improve the current results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Introduction

The notations that are used in this project are listed unless otherwise specified. We let $\mathbb{R}^{n\times m}$ be the set of all $n \times m$ real matrices. For the case when $n = m$ we denote $\mathbb{R}^{n\times n}$ the set of all $n \times n$ real matrices. For the all matrix $A \in \mathbb{R}^{n\times m}$, we let A^T be the *transpose matrix* of A. $A = A^T$ implies that A is symmetric.

When there exists an invertible matrix, S such that $U = S^{-1}VS$, then $n \times n$ matrices U and V are similar. The *null space* of a matrix A is the set of vectors $\boldsymbol{B}$ where $A\boldsymbol{B} = 0$. A matrix A is a *non-singular matrix* when its determinant is not equal to zero which also implies that the inverse matrix of A exists. When A is a singular matrix, the determinant of A is equal to zero and hence the inverse matrix of A does not exist. *Sparse matrix* is a matrix where the number of zero entries is more than the number of nonzero entries in the matrix.

Diagonal matrix is a matrix which the entries except the main diagonal are all zeros elsewhere. It is possible that the main diagonal entries to take the value zero. Thus, an $n \times m$ matrix $A = (a_{ij})$ is diagonal if:

$$a_{ij} = 0 \text{ if } i \neq j \text{ for all } i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}.$$

Let $A = [a_{ij}] \in \mathbb{R}^{n\times m}$ be a diagonal matrix. Then,

(i) when $n = m$, we have $A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 & 0 \\ 0 & a_{22} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1,n-1} & 0 \\ 0 & 0 & \cdots & 0 & a_{nn} \end{bmatrix}$.

(ii) when $n < m$, we have $A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & a_{nm} & 0 & \cdots & 0 \end{bmatrix}$.

(iii) when $n > m$, we have $A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ 0 & 0 & \cdots & a_{mm} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$.

The *identity matrix* or *unit matrix* is a square matrix with ones at the main diagonal and zeros elsewhere. Its notation is denoted as I_n or I when the size n is not important to show. An *orthogonal matrix* is a real square matrix A when the transpose of A, A^T is equal to its inverse A^{-1} that is:

$$A^TA = AA^T = I \implies A^T = A^{-1}.$$

When a square matrix A is a *normal matrix* if it commutes with its transpose A^T, that is:

$$AA^T = A^TA$$

When a matrix is an orthogonal matrix, this implies that the matrix is also a normal matrix, but a normal matrix is not necessarily an orthogonal matrix. If P is an orthogonal matrix and $B = PAP^T$, then B is orthogonally similar to A.

1.2 General Information

In the study of linear algebra, matrix factorization is a technique of splitting a matrix into 2 or more matrices. There are many different types of matrix factorization techniques and each with its own properties. For example, LU factorization is a matrix factorization technique that splits an $n \times n$ square matrix A into two matrices namely a lower triangular matrix L and an upper triangular matrix U where all the elements in matrix A, matrix L and matrix U can be real or complex numbers. Below is an example of an LU factorization of a 4×4

matrix:

$$\begin{bmatrix} 7 & 6 & -0.5 & 1 \\ 3.5 & -6 & 7.75 & 20.5 \\ -35 & -120 & 84 & 202 \\ 15.75 & -49.5 & 50.375 & 131.25 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ -5 & 10 & 1 & 0 \\ 2.25 & 7 & -3 & 1 \end{bmatrix} \begin{bmatrix} 7 & 6 & -0.5 & 1 \\ 0 & -9 & 8 & 20 \\ 0 & 0 & 1.5 & 7 \\ 0 & 0 & 0 & 10 \end{bmatrix}.$$

LU factorization is a popular factorization technique and it is easy to implement.

Other than square matrix, rectangular matrix is another form of matrix that appear frequently in real-world application. A rectangular matrix is an $m \times n$ matrix, where m can be larger than n or n can be larger than m. There are some matrix factorization techniques that are used in solving rectangular matrices such as Singular Value Decomposition (SVD), QR decomposition, rank factorization and many more.

Brownlee (2018a) stated that some computers are not able to solve large matrix efficiently. In order to solve it, matrix factorization is introduced as it reduces the large matrix into 2 or more simpler matrices that make the computation easier and increase the processing speed of the computation. In mathematics, matrix factorization is used to solve a system of linear equations, whereas in computer science, matrix factorization is used in compressing image, recommendation system, text extraction and many other applications in different fields. Beside solving complex problem, matrix factorization is also used in finding the determinant and the inverse of a matrix. Thus, knowing only one type of matrix factorization technique is not enough to solve the real-world problem.

1.3 Problem Statement

In this project, we focus on applying matrix factorization to solve real-world problems. Before we apply matrix factorization, we need to ask ourselves that "How matrix factorization techniques are carried out?". This is important as some problems can be solved effectively by using certain type of matrix factorization. This also shows us that each matrix factorization has a different structure and properties. For example, Non-Negative Matrix Factorization can be used when the values in each entry of a matrix is positive. Next, we also can ask ourselves that "Why are matrix factorization techniques are used in various applications?". By solving this problem statement, we can perform matrix factorization by using Python. Besides, we can learn the disadvantages and advantages of matrix factorization techniques to solve a particular problem. Thus, we need to construct a matrix from the problem that we want to solve and study the properties of the constructed matrix to choose a suitable matrix factorization technique.

1.4 Objective

In this project, we study the structure of matrix factorization and the applications in text extraction and other applications.

The first objective of the project is to identify different types of the matrix factorization techniques and its properties. The properties of a matrix factorization help us to determine whether the matrix factorization techniques can be used to solve the problems by using a simpler method. As an example, Cholesky Factorization and Completely Positive matrix are not commonly used in solving linear equations as both matrix factorization techniques require the original matrix to be a symmetric matrix where $A = A^T$

The second objective is to obtain relations between different types of matrix factorization techniques used in information extraction and other applications. We can use the accuracy and the time taken to measure the efficiency of the matrix factorization techniques used in information extraction and other applications. For example, application of matrix factorization techniques in solving a system of 1000 linear equations with 1000 unknowns in second:

Table 1.1: (Ng and Tan (2021)) Computation time for solving system of 1000 linear equations with 1000 unknowns in second.

Types of factorization	Real	Hermitian	Complex	Complexity
LU	0.0210	0.0198	0.0801	$O(\frac{2}{3}n^3)$
QR	0.1344	0.1446	0.3619	$O(\frac{4}{3}n^3)$
Inverse	0.0866	0.0935	0.2268	$O(n^3)$
Cholesky	N/A	0.0152	N/A	$O(\frac{1}{3}n^3)$
SVD	0.6369	0.6240	1.5364	N/A

According to Ng and Tan (2021), we see that LU Factorization has used the least amount of time to solve the linear system as compared to other matrix factorization techniques in Table 1.1.

1.5 Scope

In this project, we investigate various types of matrix factorization such as Singular Value Decomposition (SVD) and Non-Negative Matrix Factorization (NMF). We study the properties of the matrix factorizations and compare their application in text extraction and other applications.

1.6 Motivation

Currently, the dimension of the matrix is getting larger and more complex as compared to the past few years. We do need to use the advantage of matrix factorization technique to reduce the dimension and make the computation and analysis process easier. Other uses of matrix factorization are image compressing and text mining. Since the data and image can be represented in matrix form.

To have a clearer thought, we look into the example of the matrix factorization technique applied in image compression. Nowadays, people send images to their friends and family. When the image exceeds the maximum bytes size we are unable to send the image. To solve this, we use Singular Value Decomposition (SVD) which reduce the image dimension. As stated by Pandey and Umrao (2019) that SVD factorization keeps the important information of the original image by using lesser memory from the computer. Let A be a $n \times m$ rectangular matrix. SVD is to factorise A as follows:

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

where the columns of U are eigenvectors of matrix AA^T, the columns of V are eigenvectors of matrix A^TA and the matrix Σ is the diagonal matrix where its diagonal entries consist the square roots of the eigenvalues of either AA^T or A^TA and the eigenvalues are arranged decreasingly on the main diagonal.

To get the compressed image, we use the following steps as show in

Figure 1.1:

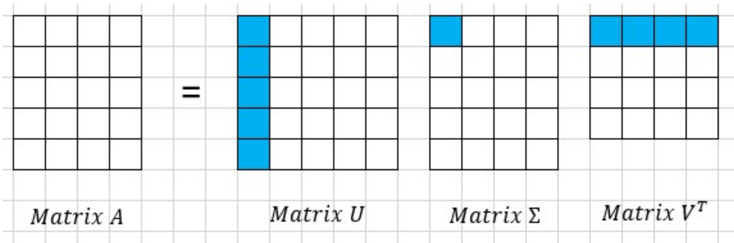

Figure 1.1: Compress image calculation when $k = 1$.

When $k = 1$, we choose the first column of matrix U, the first diagonal value of the matrix Σ and the first row of matrix V. Then we perform multiplication to get the compressed image of matrix A. We tune the value of k where $0 < k < \text{rank}(A) = \min(n, m)$, until the image is visible without losing important information.

In general, matrix factorization helps us in reducing complicated problems to a simple problem that is easy to solve. People should learn and appreciate that matrix factorization helps to ease the process of analysing. From this project, one can understand the use of matrix factorization in text extraction and other applications.

1.7 Methodology

At the beginning of the project, we collect journals and articles concerning matrix factorization techniques. After extensive reading, we study how the factorization can be implemented by using the Python.

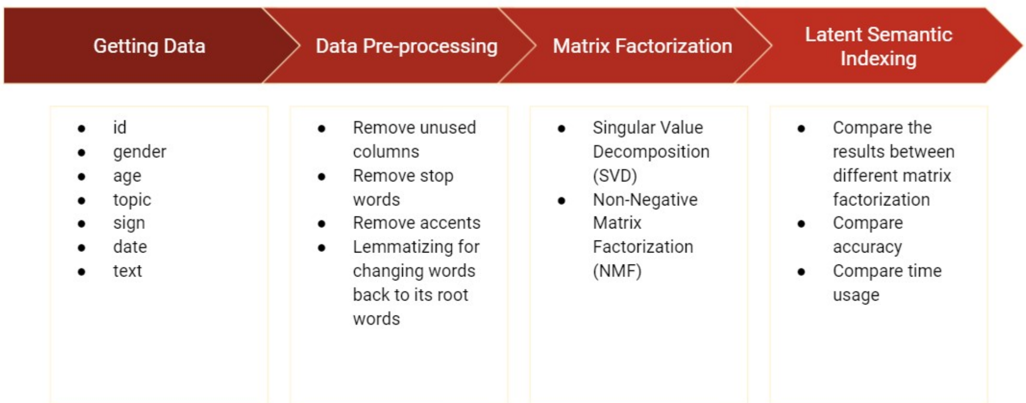

Figure 1.2: Flow chart to code.

From Figure 1.2, the first step to code is to find a data set. Then, we proceed to the second step where we remove unused columns, stop words and accents. Beside that, we also apply lemmatization which is a process to change words back to their corresponding words. For example, “went”, “going” and “goes” are changed to “go”. Next, we proceed to the third step where we apply matrix factorization to get the approximated matrix. Finally, we use the approximated matrix with Latent Semantic Indexing (LSI) to get the results. After the result is produced, we can compare the accuracy and time usage between two matrix factorizations.

1.8 Schedule

The duration in this project last for two long trimesters. Figure 1.2 shows the duration needed to complete Project 1 in January 2021 Trimester whereas Figure 1.3 shows the duration needed to complete Project 2 in May 2021 Trimester.

Table 1.2: Gantt Chart Final Year Project 1.

Week	1	2	3	4	5	6	7	8	9	10	11	12	13
Find title and supervisor	■												
Literature Review		■	■	■	■	■	■	■	■	■	■	■	■
Biweekly Report			■		■		■		■		■		■
Proposal				■	■	■	■						
Interim Report								■	■	■	■	■	
Mock Presentation						■	■						
Oral Presentation													■

Table 1.3: Gantt Chart Final Year Project 2.

Week	1	2	3	4	5	6	7	8	9	10	11	12	13
Literature Review	■	■	■	■	■	■	■	■	■	■			
Submit Mid-Semester Monitoring Form							■						
Continue writing report	■	■	■	■	■	■	■	■	■	■			
Submit draft report										■			
Submit Poster and Presentation and Presentation											■		
Submit all the form												■	
Oral Presentation													■

CHAPTER 2

LITERATURE REVIEW

2.1 Matrix Factorization

2.1.1 Singular Value Decomposition (SVD)

According to Stewart (1993), SVD was discovered in two different approaches. In the approach of linear algebra, it was founded by Eugenio Beltrami, Camille Jordan and James Joseph Sylester. The other approach is in integral equation, which was founded by Erhard Schmidt and Herman Weyl.

Theorem 2.1.1 (Horn and Johnson (2013))**.** *Let $A \in \mathbb{R}^{n\times m}$, $p = \min(n, m)$ and $\operatorname{rank}(A) = r$, then*

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

where $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{m\times m}$ are orthogonal matrices and a square diagonal matrix:

$$\Sigma_p = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_p \end{bmatrix},$$

such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 = \sigma_{r+1} = \cdots = \sigma_p$ in which:

(i) $\Sigma = \Sigma_p$ *if* $m = n$,

(ii) $\Sigma = \begin{bmatrix} \Sigma_p & 0 \end{bmatrix} \in \mathbb{R}^{n\times m}$ *if* $m > n$,

(iii) $\Sigma = \begin{bmatrix} \Sigma_p \\ 0 \end{bmatrix} \in \mathbb{R}^{n\times m}$ *if* $m < n$.

<u>Proof:</u>

Case 1: $n = m$.

Let $A_1 = A^T A$ and $A_2 = AA^T$ where $A_1, A_2 \in \mathbb{R}^{n\times n}$. Then A_1 and A_2 are symmetric matrices, this means that A_1 and A_2 have the same eigenvalues, so they are orthogonally by similar. Let S be an orthogonal matrix, then we have $A^T A = S(AA^T)S^T$ that is:

$$(SA)^T(SA) = A^T S^T SA = A^T A = S(AA^T)S^T = (SA)(SA)^T.$$

Hence, SA is a normal matrix.

Let $\lambda_1 = |\lambda_1|,\ \lambda_2 = |\lambda_2|,\ \ldots,\ \lambda_n = |\lambda_n|$ be the eigenvalues of SA, that is $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Let $\operatorname{rank}(A) = \operatorname{rank}(SA) = r$, where r is the number of nonzero eigenvalues in SA. So that $|\lambda_r| > 0$ and $|\lambda_{r+1}| = |\lambda_{r+2}| \cdots = |\lambda_n| = 0$. Let $\Sigma_p = \text{diag}\left(|\lambda_1|, |\lambda_2|, \ldots, |\lambda_n|\right)$. Let X be an orthogonal matrix, that is:

$$\begin{aligned} SA &= X\Sigma_p X^T \\ \implies A &= S^{-1}X\Sigma_p X^T \\ \implies A &= (S^{-1}X)\Sigma_p(X^T) \\ \implies A &= U\Sigma_p V^T, \end{aligned}$$

where $U = S^{-1}X$ and $V^T = X^T$ are orthogonal matrices, where the inverse of an orthogonal matrix and a product of two orthogonal matrices are still orthogonal matrices. Let the diagonal entries of Σ_p which are $\sigma_i = |\lambda_i|$ for all $i = 1, 2, \ldots, r$.

Case 2: $n < m$.

We have rank, $r \leq n$. There exists a null space of A with dimension $n \times (m-n)$. Let $X_2 = \begin{bmatrix} \boldsymbol{x_1} & \boldsymbol{x_2} & \cdots & \boldsymbol{x_{m-n}} \end{bmatrix} \in \mathbb{R}^{n\times(m-n)}$ where $\boldsymbol{x_i}$ for all $i = 1, 2, \ldots m-n$ are the set of orthonormal vectors in the null space of A. Let $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix} \in \mathbb{R}^{n\times m}$ be the orthogonal matrix, that is:

$$\begin{aligned} AX &= \begin{bmatrix} AX_1 & AX_2 \end{bmatrix} \\ \implies AX &= \begin{bmatrix} AX_1 & A\begin{bmatrix} \boldsymbol{x_1} & \boldsymbol{x_2} & \cdots & \boldsymbol{x_{m-n}} \end{bmatrix} \end{bmatrix} \\ \implies AX &= \begin{bmatrix} AX_1 & 0 \end{bmatrix}. \end{aligned}$$

We have $AX_1 \in \mathbb{R}^{n\times n}$. From Case 1, we have $AX_1 = U\Sigma_n V^T$ where $U, V \in$

$\mathbb{R}^{n\times n}$ are orthogonal matrices and $\Sigma_n = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$. Then, we have

$$\begin{aligned} A &= \begin{bmatrix} AX_1 & 0 \end{bmatrix} X^T \\ &= \begin{bmatrix} U\Sigma_n V^T & 0 \end{bmatrix} X^T \\ &= U \begin{bmatrix} \Sigma_n & 0 \end{bmatrix} \begin{bmatrix} V^T & 0 \\ 0 & I_{m-n} \end{bmatrix} X^T \\ &= U_1 \Sigma V_1^T, \end{aligned}$$

where $U_1 = U$, $\Sigma = \begin{bmatrix} \Sigma_n & 0 \end{bmatrix}$ and $V_1^T = \begin{bmatrix} V^T & 0 \\ 0 & I_{m-n} \end{bmatrix} X^T$.

Case 3: $n > m$.

We have rank, $r \leq m$. There exists a null space of A with dimension $(n-m) \times m$. Let $X_2 = \begin{bmatrix} \boldsymbol{x_1} & \cdots & \boldsymbol{x_{n-m}} \end{bmatrix}^T \in \mathbb{R}^{(n-m)\times m}$ be the orthogonal matrix, that is:

$$\begin{aligned} AX &= \begin{bmatrix} AX_1 \\ AX_2 \end{bmatrix} \\ \implies AX &= \begin{bmatrix} AX_1 \\ A \begin{bmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_{n-m}} \end{bmatrix} \end{bmatrix} \\ \implies AX &= \begin{bmatrix} AX_1 \\ 0 \end{bmatrix}. \end{aligned}$$

We have $AX_1 \in \mathbb{R}^{m\times m}$. From case 1, we have $AX_1 = U\Sigma_m V^T$ where $U, V \in \mathbb{R}^{m\times m}$ are orthogonal matrices and Σ_m is denoted as $\Sigma_m = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_m)$.

Then, we have

$$\begin{aligned} A &= \begin{bmatrix} AX_1 \\ 0 \end{bmatrix} X^T \\ &= \begin{bmatrix} U\Sigma_m V^T \\ 0 \end{bmatrix} X^T \\ &= U \begin{bmatrix} \Sigma_m \\ 0 \end{bmatrix} \begin{bmatrix} V^T & 0 \\ 0 & I_{n-m} \end{bmatrix} X^T \\ &= U_2 \Sigma V_2^T, \end{aligned}$$

where $U_2 = U$, $\Sigma = \begin{bmatrix} \Sigma_m \\ 0 \end{bmatrix}$ and $V_2^T = \begin{bmatrix} V^T & 0 \\ 0 & I_{n-m} \end{bmatrix} X^T$.

Next, to find the eigenvalues. We use the factorization of $A = U\Sigma V^T$. We know that $\mathrm{rank}(A) = \mathrm{rank}(\Sigma)$ as U and V are non-singular matrices. Now we can calculate as follows:

$$\begin{aligned} AA^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\ &= U\Sigma V^t V \Sigma^T U^T \\ &= U\Sigma\Sigma^T U^T. \end{aligned}$$

We can say that AA^T is orthogonally similar to $\Sigma\Sigma^T$.

If $n = m$, then $\Sigma\Sigma^T = \Sigma_p^2 = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2)$. If $n > m$, then $\Sigma\Sigma^T = \begin{bmatrix} \Sigma_m & 0 \end{bmatrix} \begin{bmatrix} \Sigma_m \\ 0 \end{bmatrix} = \Sigma_m^2$. Finally, $n < m$, then $\Sigma\Sigma^T = \begin{bmatrix} \Sigma_n \\ 0 \end{bmatrix} \begin{bmatrix} \Sigma_n & 0 \end{bmatrix} = \begin{bmatrix} \Sigma_n^2 & 0 \\ 0 & 0_{m-n} \end{bmatrix}$. Each of the cases, the nonzero eigenvalues of AA^T are $\sigma_1^2, \sigma_2^2, \ldots, \sigma_r^2$.

Example 2.1.2.

$$\text{Let } A = \begin{bmatrix} 1 & 3 & 4 \\ 5 & 7 & 9 \\ 11 & 3 & 1 \\ 4 & 5 & 7 \end{bmatrix}.$$

Next we find matrix AA^T and matrix A^TA

$$AA^T = \begin{bmatrix} 26 & 62 & 24 & 47 \\ 62 & 155 & 85 & 118 \\ 24 & 85 & 131 & 66 \\ 47 & 118 & 66 & 90 \end{bmatrix}, A^TA = \begin{bmatrix} 163 & 91 & 88 \\ 91 & 92 & 113 \\ 88 & 113 & 147 \end{bmatrix}.$$

Columns of U are eigenvectors of matrix AA^T, Columns of V are eigenvectors of matrix A^TA and matrix Σ is the square roots of the eigenvalues of both AA^T and A^TA. Therefore, we have:

$$\begin{bmatrix} 1 & 3 & 4 \\ 5 & 7 & 9 \\ 11 & 3 & 1 \\ 4 & 5 & 7 \end{bmatrix}$$
$$= \begin{bmatrix} -0.25 & -0.26 & 0.50 & 0.79 \\ -0.67 & -0.34 & 0.36 & -0.56 \\ -0.48 & 0.87 & 0.07 & 0.09 \\ -0.51 & -0.25 & -0.79 & 0.25 \end{bmatrix} \begin{bmatrix} 18.17 & 0 & 0 \\ 0 & 8.46 & 0 \\ 0 & 0 & 0.36 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.60 & 0.78 & -0.18 \\ -0.52 & -0.21 & 0.83 \\ -0.61 & -0.59 & 0.53 \end{bmatrix}^T.$$

2.1.2 Non-Negative Matrix Factorization (NMF)

Definition 1 (Zurada et al. (2013))**.** Let $A \in \mathbb{R}^{n\times m}$ be an $n \times m$ matrix such that every element in matrix A is positive, factorize A into matrix $W \in \mathbb{R}^{n\times r}$ and matrix $H \in \mathbb{R}^{r\times m}$ with $r \leq \min(m, n)$ such that:

$$A \approx WH.$$

where the entries in matrices W and H are non-negative. This factorization is called Non-Negative Matrix Factorization (NMF).

There are many applications of NMF. For example, NMF is used in face recognition, bioinformatics, text mining and audio (speech) recognition. There are many different methods to calculate the matrices W and H. Each of the methods produces different W and H.

According to Zurada et al. (2013), the most commonly used method is

Euclidean distance measure starting with random initialisation of the values of W and H. Multiplicative update rules are used to minimize the error function as below:

$$H_{ij} \leftarrow H_{ij} \frac{(W^T A)_{ij}}{(W^T W H)_{ij}},$$
$$W_{ij} \leftarrow W_{ij} \frac{(A H^T)_{ij}}{(W H H^T)_{ij}}.$$

The multiplicative update rules can prevent the error function from increasing and when the last 4 decimal figures of the last two iterations are the same then we stop the iterations. Another method is by projecting gradient descent and alternating least squares as this method converges in a faster pace.

Example 2.1.3. Let A be a 6×3 positive matrix as follows which is randomly generated by using Python:

$$A = \begin{bmatrix} 5 & 18 & 14 \\ 16 & 6 & 18 \\ 2 & 18 & 2 \\ 3 & 13 & 17 \\ 6 & 3 & 5 \\ 0 & 6 & 10 \end{bmatrix}.$$

We then use Python to approximate matrix W and matrix H:

$$W = \begin{bmatrix} 1.72 & 1.76 \\ 4.35 & 0 \\ 0 & 1.81 \\ 2.15 & 1.26 \\ 1.34 & 0.09 \\ 0.60 & 1.74 \end{bmatrix} \text{ and } H = \begin{bmatrix} 3.23 & 1.25 & 4.49 \\ 0 & 9.11 & 3.33 \end{bmatrix}.$$

Hence we have the factorization as below:

$$\begin{bmatrix} 5 & 18 & 14 \\ 16 & 6 & 18 \\ 2 & 18 & 2 \\ 3 & 13 & 17 \\ 6 & 3 & 5 \\ 0 & 6 & 10 \end{bmatrix} \approx \begin{bmatrix} 1.72 & 1.76 \\ 4.35 & 0 \\ 0 & 1.81 \\ 2.15 & 1.26 \\ 1.34 & 0.09 \\ 0.60 & 1.74 \end{bmatrix} \begin{bmatrix} 3.23 & 1.25 & 4.49 \\ 0 & 9.11 & 3.33 \end{bmatrix}.$$

2.2 Application of Matrix Factorization in Text Extraction

2.2.1 Latent Semantic Indexing (LSI)

In this section, we look into Latent Semantic Indexing which is an information retrieval. It is to retrieve the document requested by the user. Vasireddy (2009) stated that there are n words and m documents in a particular database, where all the common words like "a", "the" , "an", "this" and many more are removed from each of the documents. Then it can be written as a matrix A where rows of A represent words and columns of A represent documents.

In Latent Semantic Indexing, SVD is used. According to Deerwester et al. (1990), SVD is used in reducing the dimension of the original matrix. The data in the original matrix contains useless data that can affect the accuracy of the text extraction algorithm. The original matrix is factored into 3 matrices by using SVD. Hence, we create another matrix that is an approximation of the original matrix which contains less useless data. Thus, the accuracy of the algorithm can be increased.

According to Vasireddy (2009), problems faced if LSI is used are synonymy and polysemy. Synonymy is defined to be a set of different words that have the same meaning and polysemy is defined to be a word that has many meanings. LSI can only be used on smaller document database, something like World Wide Web is not applicable.

2.2.2 Hyper-Link Induced Topic Search

Kleinberg (1999) developed the search algorithm name Hyper-Link Induced Topic Search (HITS) and the title of the article is Authoritative Sources in a

Hyperlinked Environment in 1997. HITS algorithm is used to extract link structure text in the World Wide Web (WWW) database.

As the number of years increases, the number of hyper-linked documents increases as well. Two new terms introduced are authority and hubs. Authority is a website that has authority to post or discuss a particular subject. For example, "www.utar.edu.my". While hubs mean a web-page that links to many related authority pages.

When we type "UTAR" on the search bar, "www.utar.edu.my" should be the most authoritative page. However, the website "www.utar.edu.my" does not use the word "UTAR" as often as other pages in WWW. We can say that most of the authority pages do not use the term frequently. Therefore, this affected the web-page does not rank the highest in the search list. As stated by Vasireddy (2009), simple text-based search engine is not workable in hyper-linked document as it finds the relevant document based on the number of appearances of the same term on that page.

Kleinberg (1999) states that the authority and hub scores are used to determine which are the pages that have good authority and hub. Given a large graph that contains vertices and lines that connect the vertices. We need to find a subgraph, G_σ which contains relevant pages based on the user input. An iterative algorithm is used to update and maintain the authority score and hub score for each of the pages. Authority and hub score are non-negative values. We continue the iteration until the score is the same with the previous iteration value and normalize the value where the sum of squares equal to 1. The better authority pages and hub pages are determined by the highest authority score and hub score.

2.2.3 Document Clustering with NMF

Zurada et al. (2013) did a report to compare different methods to calculate NMF which are Euclidean distance and corr-entropy. 20-newsgroup data set, which is a popular data set in text clustering and classification. This data set contains about 20,000 documents and 20 different newsgroups.

Entropy measure is to evaluate clustering performance. Below is the formula for total entropy. Let A be the total entropy for a set of clusters, k be the weighted mean of the entropies of each cluster weighted and 1 is the size of each cluster:

$$A = \frac{k}{l}.$$

First, the value of distribution for each cluster data is calculated. Let p_{ij} be the probability cluster i belong to class j:

$$p_{ij} = \frac{m_{ij}}{m_i},$$

where m_{ij} is the number of class i in cluster j and m_i is the number of elements in cluster i. Next, the value for entropy of each cluster i is calculated using the formula which is shown below:

$$e_i = -\sum_{j=1}^{L} p_{ij} \log_2(p_{ij}),$$

where L is the total number of classes. Final step, the value for entropy is calculated using the formula which is shown as below:

$$e = \sum_{i=1}^{K} \frac{m_i}{m} e_i.$$

Then result in Table 2.1 is obtained:

Table 2.1: (Zurada et al. (2013)) Entropy of 20-Newsgroups data set with NMF-PGD(EucD) and NMF-Corr.

Number of Clusters (k)	NMF-PGD (EucD)	NMF-Corr ($\sigma = 1$)	NMF-Corr ($\sigma = 0.5$)	NMF-Corr ($\sigma = 0.01$)
$r = 2$	3.84	3.86	3.85	4.30
$r = 3$	3.86	3.79	3.58	4.27
$r = 4$	3.78	3.49	3.50	4.27
$r = 5$	3.74	3.60	3.38	4.24
$r = 6$	3.49	3.36	3.30	4.23
$r = 7$	3.44	3.28	3.26	4.20
$r = 8$	3.30	3.26	2.94	4.19
$r = 9$	3.30	3.34	3.13	4.18
$r = 10$	3.16	3.23	2.93	4.20

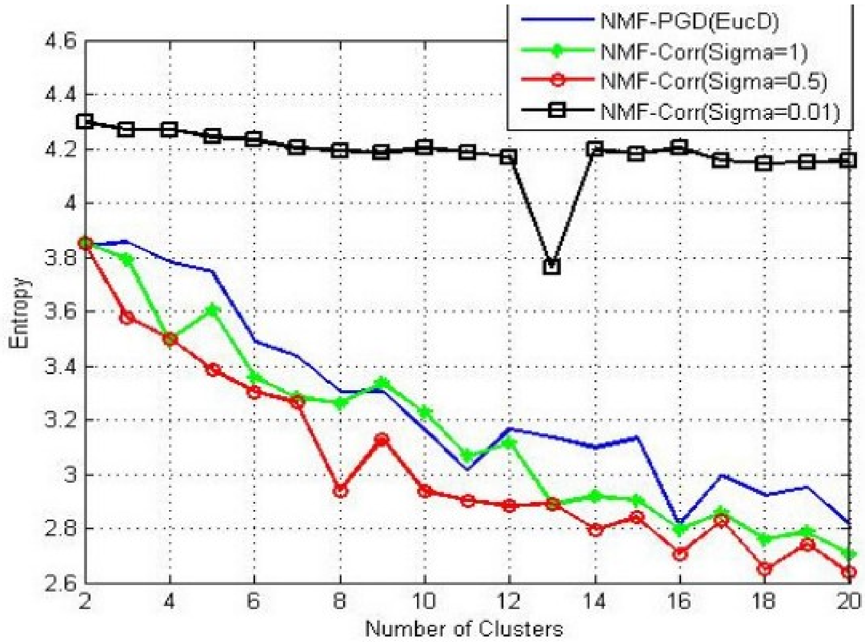

Figure 2.1: (Zurada et al. (2013)) Entropy comparison for NMF-PGD(EucD) and NMF-Corr.

When we look into the entropy values, the lowest entropy has better clustering performance. From Table 2.1 and Figure 2.1 we observe that NMF-Corr ($\sigma = 0.5$) has lower entropy value as compared to NMF-PGD (EucD). NMF-Corr means that the matrices of W and H are calculated by using Corr-entropy similarity measure and the formula is shown as below:

$$\text{Corr}\,(A, WH) = \sum_{i,j} \exp\left(\frac{-(A_{ij} - (WH)_{ij})^2}{2\sigma^2}\right),$$

where σ act as a parameter for corr-entropy similarity measure.

According to Ensari, Choroski and Zurada (2012), Corr-entropy is a localized similarity measure between two random variables. Thus it is used as a objective function for NMF to compute the similarity between the original matrix and approximated matrix. Based on the goal of the given problems, the objective function is either maximized or minimized.

2.2.4 Discovering Relations using Matrix Factorization Methods

As stated by Cergani and Miettinen (2013), the main purpose of information extraction is to extract facts from the free-form text in the Web database. Free-form text is a text that has no fixed form. In an old-fashioned information extraction, we ourselves need to define the extraction rules or training example that the user is interested in. Therefore, Cergani and Miettinen (2013) introduced the method of matrix factorization, that is, Non-negative Matrix Factorization (NMF) and Boolean Matrix Factorization (BMF).

In the approach of NMF, they first build a context-by-context co-occurrence matrix O for each pair of (C_1, C_2) where C_1 and C_2 are the category of words. $O(i, j)$ contain the number of Noun Entity (NE) pairs. For example, NE that has the teacher-school relation are grouped together. Next, normalizing the matrix O so that the sum of each row is 1, where each row is divided by non-zero value in the particular row. Then both non-negative co-occurrence matrix O and integer, k are calculated. Next, finding the non-negative matrices W and H of k rows and columns, minimizing objective function, $||O - WH||_F^2$. Columns of matrix W are the raw candidate relations as it gives us the non-negative weights between each context and candidate relations. In order to get the final candidate cluster, rounding matrix W to binary and if $W(i, j) = 1$, then context i is the candidate relations of j.

The next approach is using BMF. We have a matrix C as a Boolean Product of binary factorization matrices A and B, where $C \approx A \circ B$. Boolean product is defined as $(A \circ B)_{i,j} = \bigvee_{k=1}^{l} (a_{i,k} \wedge b_{k,j})$, where $\vee$ is the logical OR operator and $\wedge$ is the logical AND operator. BMF is used in minimizing the Hamming Distance between C and $A \circ B$ that is the number of places where C and $A \circ B$ differ. Matrix C is the context patterns-by-instance pairs, matrix A is the context patterns to candidate relation and matrix B is the instance pairs to candidates relation. At the end, context corresponding row of C which is closest to the k^{th} row of B is selected as the relation's name.

CHAPTER 3

PRELIMINARY RESULTS

In this chapter, two examples are presented as the preliminary results of the two methods mentioned in literature review, which are Latent Semantic Indexing (LSI) and Hyper-Link Induced Topic Search (HITS). Each of the examples is computed with two methods. Matrix factorization techniques is not used in one of the methods whereas the other is using matrix factorization techniques. The aim of using two methods is to show that the matrix factorization gives the same results as the method that does not use matrix factorization.

3.1 Latent Semantic Indexing (LSI)

Suppose there are three documents, as follows:

- (i) Document 1: I see a cat. That cat sat.
- (ii) Document 2: I can pat the cat.
- (iii) Document 3: I have a mat.

Next, we calculate the number of the same word appear in each document. Then, we convert the information obtained to a matrix in such a way that is shown in Table 3.1

Table 3.1: Conversion of documents to a matrix.

	Document 1	Document 2	Document 3
I	1	1	1
see	1	0	0
cat	2	1	0
sat	1	0	0
can	0	1	0
pat	0	1	0
have	0	0	1
mat	0	0	1

Then, we obtain

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

that represents the 3 documents.

After A is found, we need an $m \times 1$ matrix y which is the input of the user to find related documents. Next, we use the formula $x = Ay$ to calculate matrix x, where x is used to rank the documents. The value of $(k, 1)$-entry of x is the number of words searched by the user that appear in document k, where $k = 1, 2, 3$.

As an example, if a user input, "I see a cat sat." The vector y has the value as in Table 3.2:

Table 3.2: Conversion of user input to a matrix.

	User input
I	1
see	1
cat	1
sat	1
can	0
pat	0
have	0
mat	0

and hence

$$y = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Then

$$x = \begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 1 \end{bmatrix}.$$

From the computation, we see that Document 1 has the highest value followed by Document 2 and Document 3. Then the order of the ranking is Document 1, Document 2 and Document 3. This ranking is presented to the user based on the similarity with the user input.

From the previous computation, we did not apply any matrix factorization techniques. The next computation, matrix factorization technique is used in reducing the dimension of the matrix A. By Theorem 2-1.1 (Horn and Johnson (2013)), let $A \in \mathbb{R}^{n\times m}$, then A can be factorized into three different matrices that is $A = U\Sigma V^T$. Then

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix},$$

$$U = \begin{bmatrix} -0.517 & -0.441 & 0 & -0.096 & -0.167 & -0.167 & -0.487 & -0.487 \\ -0.272 & 0.232 & 0.289 & -0.386 & 0.554 & 0.554 & -0.112 & -0.112 \\ -0.717 & 0.318 & 0 & -0.194 & -0.242 & -0.242 & 0.339 & 0.339 \\ -0.272 & 0.232 & 0.289 & 0.87 & 0.096 & 0.096 & -0.08 & -0.08 \\ -0.172 & -0.147 & -0.577 & 0.145 & 0.704 & -0.296 & 0.074 & 0.074 \\ -0.172 & -0.147 & -0.577 & 0.145 & -0.296 & 0.704 & 0.074 & 0.074 \\ -0.072 & -0.526 & 0.289 & 0.048 & 0.083 & 0.083 & 0.743 & -0.257 \\ -0.072 & -0.526 & 0.289 & 0.048 & 0.083 & 0.083 & -0.257 & 0.743 \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} 3.027 & 0 & 0 \\ 0 & 1.685 & 0 \\ 0 & 0 & 1.414 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$V^T = \begin{bmatrix} -0.825 & -0.522 & -0.218 \\ 0.391 & -0.248 & -0.886 \\ 0.408 & -0.816 & 0.408 \end{bmatrix}.$$

Matrices U, Σ and V^T are computed and we reconstruct matrix A. Let U_k be the matrix that contains the first k columns of U, let Σ_k be the sub-matrix that contains the first k rows and the first k columns of Σ where the diagonal entries are square roots of the corresponding eigenvalues of AA^T or A^TA and let V_k^T be the matrix that contains the first k rows of V^T where $k = 1, 2, 3$. First, we take $k = 1$, we have:

$$U_1 = \begin{bmatrix} -0.517 \\ -0.272 \\ -0.717 \\ -0.272 \\ -0.172 \\ -0.172 \\ -0.072 \\ -0.072 \end{bmatrix}, \ \Sigma_1 = \begin{bmatrix} 3.027 \end{bmatrix} \text{ and } V_1^T = \begin{bmatrix} -0.825 & -0.522 & -0.218 \end{bmatrix}.$$

We construct the matrix A using the matrices above, that is:

$$A = \begin{bmatrix} -0.517 \\ -0.272 \\ -0.717 \\ -0.272 \\ -0.172 \\ -0.172 \\ -0.072 \\ -0.072 \end{bmatrix} \begin{bmatrix} 3.027 \end{bmatrix} \begin{bmatrix} -0.825 & -0.522 & -0.218 \end{bmatrix}$$

$$= \begin{bmatrix} 1.291 & 0.817 & 0.341 \\ 0.679 & 0.43 & 0.179 \\ 1.791 & 1.133 & 0.473 \\ 0.679 & 0.43 & 0.179 \\ 0.43 & 0.272 & 0.113 \\ 0.43 & 0.272 & 0.113 \\ 0.18 & 0.114 & 0.048 \\ 0.18 & 0.114 & 0.048 \end{bmatrix}.$$

Then, by using the user's input matrix y, we have

$$x = \begin{bmatrix} 1.291 & 0.679 & 1.791 & 0.679 & 0.43 & 0.43 & 0.18 & 0.18 \\ 0.817 & 0.43 & 1.133 & 0.43 & 0.272 & 0.272 & 0.114 & 0.114 \\ 0.341 & 0.179 & 0.473 & 0.179 & 0.113 & 0.113 & 0.048 & 0.048 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 4.44 \\ 2.81 \\ 1.172 \end{bmatrix}.$$

Next, we repeat the process for $k = 2, 3$. Then the following table is constructed:

Table 3.3: Comparison of methods.

	Basic Method	SVD $(k = 1)$	SVD $(k = 2)$	SVD $(k = 3)$
Document 1	5	4.44	4.665	4.999
Document 2	2	2.81	2.667	1.999
Document 3	1	1.172	0.664	0.998

From Table 3.3, we observe that Document 1 has the highest value as compared to the other documents. Therefore, a ranking list of Document 1 followed by Document 2 and then Document 3 is presented to the user which are highly compatible with the user's input.

3.2 Hyper-Link Induced Topic Search (HITS)

The implementation steps of HITS algorithm are shown as follows:

(i) We need to prepare a World Wide Web data set.

(ii) We convert the data set to a finite number of vertices and directed edges connecting the vertices.

(iii) User input is required.

(iv) A sub-graph is extracted from the data set and contain what the user wants to search.

(v) An adjacency matrix A is obtain from the sub-graph.

(vi) Two methods are used to calculate the ranking. Method 1 is not using matrix factorization techniques while method 2 is using matrix factorization techniques.

(vii) We used method 1 to cross-check with method 2 to see if the same result is produced.

(viii) The ranking list is generated.

World Wide Web data set can be represented in a graph with finite number of vertices and directed edges connecting the vertices. This is also known as a directed graph as the edges show the direction from one vertex to another vertex. Vertices in the graph represent websites. The directed edges represent links from a website to another website. After a directed sub-graph, which contain user's input is obtained such as Figure 3.1.

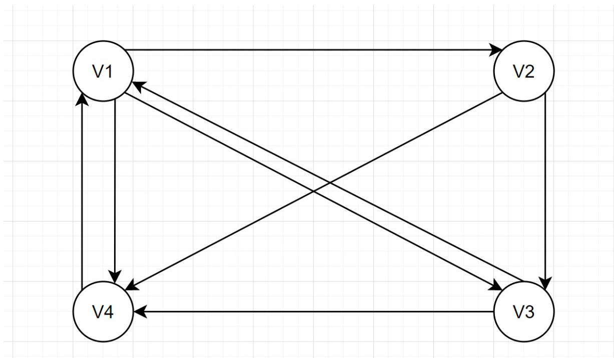

Figure 3.1: Sub-graph extracted from World Wide Web data set.

We obtain the adjacency matrix A from the sub-graph above

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Then,

$$A^T = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}.$$

Method 1 is used to calculate the scores without applying matrix factorization techniques. Let $\boldsymbol{u}$ be the hub scores vector and $\boldsymbol{v}$ be the authority scores vector where hub score is the sum of the authority scores of each node that is pointed to it and authority score is the sum of the hub scores of each node that is pointed to it. We apply

$$\boldsymbol{v_k} = A^T \boldsymbol{u_{k-1}} \text{ and } \boldsymbol{u_k} = A\boldsymbol{v_k},$$

where k is the number of iterations.

First, we choose an initial vector, $\boldsymbol{u_0} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$. When $k = 1$,

$$\boldsymbol{v_1} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 2 \\ 3 \end{bmatrix},$$

$$\boldsymbol{u_1} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \\ 5 \\ 2 \end{bmatrix}.$$

We construct the following table from $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$.

Table 3.4: Hub and Authority scores when $k = 1$.

$k = 1$	Hub Scores	Authority Scores
V1	6	2
V2	5	1
V3	5	2
V4	2	3

Next, we calculate the new authority scores by using the value from $k = 1$:

$$\boldsymbol{u_2} = \frac{1}{\sqrt{6^2 + 5^2 + 5^2 + 2^2}} \begin{bmatrix} 6 \\ 5 \\ 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.63246 \\ 0.52705 \\ 0.52705 \\ 0.21082 \end{bmatrix},$$

$$\boldsymbol{v_2} = \frac{1}{\sqrt{2^2 + 1^2 + 2^2 + 3^2}} \begin{bmatrix} 2 \\ 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.4714 \\ 0.2357 \\ 0.4714 \\ 0.70711 \end{bmatrix}.$$

We continue to construct the following table

Table 3.5: Hub and Authority scores when $k = 2$.

$k = 2$	Hub Scores	Authority Scores
V1	0.63246	0.4714
V2	0.52705	0.2357
V3	0.52705	0.4714
V4	0.21083	0.70711

Since the value of hub scores and authority scores in Table 3.4 and Table 3.5 are different. Therefore, we need to calculate the value for $k = 3$. Then we have the result as below:

Table 3.6: Hub and Authority scores when $k = 3$.

$k = 3$	Hub Scores	Authority Scores
V1	0.63246	0.4714
V2	0.52705	0.2357
V3	0.52705	0.4714
V4	0.21083	0.70711

When, we get the same hub scores and authority scores from Table 3.5 and Table 3.6. We stop the iteration. From the result, we can see that V1 is a good hub as the score is the highest and V4 is a good authority as the score is the highest. A good hub means that it links to many other websites, whereas a good authority means that many websites are linked to it. V1 and V4 rank the highest in the ranking list.

Next, Method 2 is used to calculate the scores by using Theorem 2-1.1 (Horn and Johnson (2013)) where $A \in \mathbb{R}^{n\times m}$, then A can be expressed in the form, $A = U\Sigma V^T$ where the columns of U are eigenvectors of matrix AA^T, the columns of V are eigenvectors of matrix A^TA and the matrix Σ is the diagonal matrix where its diagonal entries consist the square roots of the eigenvalues of either AA^T or A^TA and the eigenvalues are arranged decreasingly on the main diagonal. Then

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

$$U = \begin{bmatrix} -0.69994 & -0.35162 & -0.61348 & 0.1004 \\ -0.56593 & -0.18516 & 0.68243 & -0.42394 \\ -0.42394 & 0.68243 & 0.18516 & 0.56593 \\ -0.1004 & 0.61348 & -0.35162 & -0.69994 \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} 2.28533 & 0 & 0 & 0 \\ 0 & 1.45341 & 0 & 0 \\ 0 & 0 & 0.68804 & 0 \\ 0 & 0 & 0 & 0.43757 \end{bmatrix},$$

$$V^T = \begin{bmatrix} -0.22944 & -0.30628 & -0.55391 & -0.73942 \\ 0.89164 & -0.24193 & -0.36933 & 0.10021 \\ -0.24193 & -0.89164 & 0.10021 & 0.36966 \\ -0.30628 & 0.22944 & -0.73942 & 0.55391 \end{bmatrix}.$$

The hub scores can be obtained from the first column in matrix U and authority scores can be obtained from the first row of V^T. We have the vectors as below.

$$\boldsymbol{u} = \begin{bmatrix} -0.69994 \\ -0.56593 \\ -0.42394 \\ -0.1004 \end{bmatrix}, \ \boldsymbol{v} = \begin{bmatrix} -0.22944 \\ -0.30628 \\ -0.55391 \\ -0.73942 \end{bmatrix}.$$

The hub scores and authority scores are used to rank the vertices. Ranking is an ordinal data where the value less than or equal to zero does not hold any meaning. So, we take the absolute value. Then, we have the following table:

Table 3.7: Calculation of hub and authority scores using SVD.

	Hub Scores	Authority Scores
V1	0.69994	0.22944
V2	0.56593	0.30628
V3	0.42394	0.55391
V4	0.1004	0.73942

Form the Table 3.7 above, we conclude that V4 is the best authority website and V1 is the best hub website. In the ranking list, V4 and V1 rank the highest. This shows both methods have the same result.

CHAPTER 4

MAIN RESULTS

4.1 Data Introduction

In this project, we the Blog Authorship Corpus data set from Kaggle created by Tatman (2017). This data set consists of 19320 bloggers from blogger.com in August 2014. We have a total of 681288 posts with over 140 million words. The blog is placed on a separate file, which consists of blogger id as name, gender, age, industry, astrological sign. The data set can be separated into 40 different categories.

In this project, as the number of words is large, we are not able to use all the data due to lack of random-access memory (RAM). Instead of taking the whole data, we take the number of posts which is denoted by n per category and so we have $40n$ posts. The number n can be increase based on the computer's RAM.

4.2 Data Setup

Before the data can be used for information extraction, we need to do some data cleaning and customization. Appendix A contains the code for customize the data and cleaning the data. In this section, we remove the punctuations in the sentences and reduce the word back to their root form respectively. For example, "goes", "went" and "going" are change to go.

After the cleaning process, we then convert the clean data and the user data into data frame where the column is the number of document and the rows is the words exists in all the document. Appendix B is the code of converting the clean data whereas Appendix C is the code of converting the user data.

In this project, we use 3 different sizes of data set which are 400, 800 and 1200 rows of data.

4.3 Latent Semantic Indexing (LSI) with Singular Value Decomposition (SVD)

In this section, SVD is used as the matrix factorization techniques. We refer the code written by Brownlee (2018b). The formula of truncated SVD is $A =$

$U_k\Sigma_k V_k^T$, where $k = 1, 2, \ldots, \mathrm{rank}(A)$. This technique uses a smaller size of U, Σ and V^T to approximate the original matrix A corresponding to the value of k. The goal is to find the best value of k.

To find the best k, we use two different objective functions. The first objective function is,

$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \approx \xi, \tag{4.1}$$

where σ_i^2 is the eigenvalue of AA^T, k is the minimum value use to approximate the original matrix, r is the total number of eigenvalues from the original matrix and ξ is the ratio. In the above function we can change the value of ξ. In this project, we let ξ to be 0.85, 0.90 and 0.95. This means that we are keeping 85%, 90% and 95% of the eigenvalues.

The second objective function is to use Frobenius norm which is,

$$||A - B||_{Fro} = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} |a_{ij} - b_{ij}|^2},$$

where matrix B is the approximated matrix from $U_k\Sigma_k V_k^T$. Both the objective functions are used for minimizing the value of k, to get a better approximated matrix of the original matrix.

Table 4.1: Result of 400 rows data.

Rank	LSI	LSI (SVD) $\xi = 0.85$	LSI (SVD) $\xi = 0.90$	LSI (SVD) $\xi = 0.95$	LSI (SVD) Frobenius Norm
1	Doc 201	Doc 201	Doc 201	Doc 201	Doc 201
2	Doc 220	Doc 220	Doc 220	Doc 220	Doc 220
3	Doc 168	Doc 303	Doc 388	Doc 168	Doc 168
4	Doc 303	Doc 388	Doc 303	Doc 303	Doc 303
5	Doc 388	Doc 168	Doc 168	Doc 359	Doc 388
6	Doc 359	Doc 359	Doc 359	Doc 388	Doc 359
7	Doc 165	Doc 306	Doc 306	Doc 165	Doc 165
8	Doc 13	Doc 13	Doc 13	Doc 13	Doc 126
9	Doc 37	Doc 165	Doc 165	Doc 306	Doc 37
10	Doc 72	Doc 230	Doc 123	Doc 72	Doc 306

Table 4.2: Result of 800 rows data.

Rank	LSI	LSI (SVD) $\xi = 0.85$	LSI (SVD) $\xi = 0.90$	LSI (SVD) $\xi = 0.95$	LSI (SVD) Frobenius Norm
1	Doc 401	Doc 401	Doc 401	Doc 401	Doc 401
2	Doc 98	Doc 154	Doc 98	Doc 98	Doc 98
3	Doc 39	Doc 430	Doc 430	Doc 430	Doc 39
4	Doc 154	Doc 98	Doc 154	Doc 154	Doc 154
5	Doc 430	Doc 768	Doc 603	Doc 328	Doc 430
6	Doc 38	Doc 603	Doc 768	Doc 768	Doc 38
7	Doc 325	Doc 606	Doc 328	Doc 603	Doc 768
8	Doc 603	Doc 23	Doc 612	Doc 709	Doc 709
9	Doc 328	Doc 574	Doc 574	Doc 612	Doc 328
10	Doc 612	Doc 767	Doc 606	Doc 38	Doc 612

Table 4.3: Result of 1200 rows data.

Rank	LSI	LSI (SVD) $\xi = 0.85$	LSI (SVD) $\xi = 0.90$	LSI (SVD) $\xi = 0.95$	LSI (SVD) Frobenius Norm
1	Doc 955	Doc 601	Doc 601	Doc 601	Doc 955
2	Doc 601	Doc 224	Doc 955	Doc 955	Doc 601
3	Doc 144	Doc 640	Doc 138	Doc 144	Doc 144
4	Doc 138	Doc 138	Doc 224	Doc 138	Doc 138
5	Doc 49	Doc 906	Doc 640	Doc 640	Doc 49
6	Doc 640	Doc 955	Doc 144	Doc 224	Doc 640
7	Doc 224	Doc 1148	Doc 903	Doc 488	Doc 224
8	Doc 48	Doc 652	Doc 1148	Doc 903	Doc 48
9	Doc 485	Doc 903	Doc 1191	Doc 1148	Doc 1042
10	Doc 488	Doc 59	Doc 906	Doc 1059	Doc 1059

In Tables 4.1 to 4.3, documents locate on rank 1 row are documents that rank the highest corresponding to the each method used. As an example, Doc 201 is the first and Doc 220 is the second in the ranking where LSI method is used in Table 4.1. The third, fourth and fifth columns from the tables above are the results calculated using equation (4.1). The sixth column is the result calculated using Frobenius norm.

From the tables above, we used the code from Appendix D to get the LSI result without applying SVD. This result is useful in determining whether the approximation matrix can obtain similar result. From the tables, when $\xi = 0.85$, the top 10 highest documents are different from LSI column in Table 4.3, only the first is the same in Table 4.2 and the first two are the same in Table 4.1. This is due to lack of data to have a better approximation of the original data. When we start to increase the ratio, ξ to 0.90, it is the same as inserting more data. Unfortunately, in Table 4.1 and Table 4.2 only the first two documents are the same as in LSI column while in Table 4.3 all are different but it is getting closer to the original result. Next, we increase the ratio, ξ to 0.95. The similarity pattern is starting to form in Tables 4.2 and 4.3, whereas Table 4.1 having top 4 to be the same. The equation (4.1) is use to get the best approximation and the code is shown in Appendix E.

After looking into third to fifth columns, we are unable to get similar result. This due to the inappropriate value of k is chosen. In order to get a better value of k we used Frobenius norm as our objective function. This method we do not need to guess the value as it directly gives the most suitable value of k. The code can be referred to in Appendix E. From the result, we get the first seven that are similar to LSI column in Table 4.1, the first six and the last two are similar in Table 4.2 and the first eight are similar in Table 4.3. From the tables, we found that this method is 70% to 80% accurate.

4.4 Latent Semantic Indexing (LSI) with Non-Negative Matrix Factorization (NMF)

In this section, we are going to discuss NMF applied in LSI. We are using NMF model from Scikit-learn.org (2019) where the objective function is shown as below:

$$0.5\,||A - WH||_{loss}^2 + \alpha l_{1_{ratio}}\,||\text{vec}(W)||_1 + \alpha l_{1_{ratio}}\,||\text{vec}(H)||_1 \\ + 0.5\alpha(1 - l_{1_{ratio}})\,||W||_{Fro}^2 + 0.5\alpha(1 - l_{1_{ratio}})\,||H||_{Fro}^2\,,$$

where

$$||A||_{Fro}^2 = \sum_{ij} A_{ij}^2 \text{ (Frobenius norm)},$$
$$||\text{vec}(A)||_1 = \sum_{ij} \text{abs}(A_{ij}) \text{ (elementwise L1 norm)}.$$

In this model, we have a few parameters that we can tune such as "init", "solver" and "beta_loss". In the parameter "init", we use two different methods to initialise the matrix W and matrix H which are non-negative random matrices, scaled with $\sqrt{\frac{X.mean()}{n\_components}}$ and non-negative double singular value decomposition as this method is better for sparseness. These methods are used to initialize both W and H matrices. Next, we look into the parameter "solver" which has two methods to update the matrix W and matrix H which are coordinates descent and multiplicative update. These methods are used to calculate new W and H matrices. Finally, we look into the parameter "beta_loss" where we use two different methods which are Frobenius norm and Kullback-Leibler divergence. These methods are used as the objective function to minimize the errors between the previous updated matrix and new updated matrix. When the value of the objective functions is stable then the loop to run the update methods stops.

Table 4.4: Parameter tuning for 400 rows data.

No	Parameter "init"	Parameter "solver"	Parameter "beta_loss"	$\|\|A - WH\|\|_{Fro}^2$
1	random	cd	Frobenius	31.84471
2	random	mu	Frobenius	44.47370
3	random	mu	Kullback-Leibler	67.52773
4	nndsvd	cd	Frobenius	33.25570
5	nndsvd	mu	Frobenius	92.91888
6	nndsvd	mu	Kullback-Leibler	238.21473

Table 4.5: Parameter tuning for 800 rows data.

No	Parameter "init"	Parameter "solver"	Parameter "beta_loss"	$\|\|A - WH\|\|_{Fro}^2$
1	random	cd	Frobenius	57.96992
2	random	mu	Frobenius	110.47584
3	random	mu	Kullback-Leibler	123.33882
4	nndsvd	cd	Frobenius	51.54955
5	nndsvd	mu	Frobenius	139.66458
6	nndsvd	mu	Kullback-Leibler	262.12321

Table 4.6: Parameter tuning for 1200 rows data.

No	Parameter "init"	Parameter "solver"	Parameter "beta_loss"	$\|\|A - WH\|\|_{Fro}^2$
1	random	cd	Frobenius	57.96992
2	random	mu	Frobenius	110.47584
3	random	mu	Kullback-Leibler	123.33882
4	nndsvd	cd	Frobenius	51.54955
5	nndsvd	mu	Frobenius	139.66458
6	nndsvd	mu	Kullback-Leibler	262.12321

In Table 4.4, Table 4.5 and Table 4.6 are different combination of parameters with their respective Frobenius norm. The lower the Frobenius norm is, the smaller the error between matrix A and matrix WH is. The code can be viewed in Appendix F. In each of the tables, we can see that combination 1 and combination 4 have a smaller Frobenius norm as compared to the other combinations.

Next, we use these combinations to approximate matrix W and matrix H and use it in LSI. The results are tabulated in Table 4.7.

Table 4.7: LSI results using NMF.

Rank	LSI 400	LSI NMF 400	LSI 800	LSI NMF 800	LSI 1200	LSI NMF 1200
1	Doc 201	Doc 201	Doc 401	Doc 401	Doc 601	Doc 601
2	Doc 220	Doc 220	Doc 98	Doc 98	Doc 955	Doc 955
3	Doc 165	Doc 168	Doc 39	Doc 39	Doc 144	Doc 144
4	Doc 168	Doc 303	Doc 154	Doc 430	Doc 138	Doc 138
5	Doc 303	Doc 388	Doc 430	Doc 154	Doc 49	Doc 224
6	Doc 359	Doc 359	Doc 38	Doc 38	Doc 224	Doc 49
7	Doc 388	Doc 165	Doc 325	Doc 709	Doc 640	Doc 640
8	Doc 13	Doc 306	Doc 328	Doc 328	Doc 48	Doc 1059
9	Doc 37	Doc 13	Doc 603	Doc 603	Doc 485	Doc 1148
10	Doc 72	Doc 37	Doc 612	Doc 612	Doc 488	Doc 912

From the above table, we can see which document ranks the highest corresponding to the method used. For example, Doc 201 is the first and Doc 220 is the second in the ranking when LSI is used with 400 data rows. From the results in Table 4.7, we can see that applying NMF in LSI did not perform well as compared to SVD. We still can see the trend as compared to the original result and the accuracy is increasing when we increase the number of data rows. In 400 data rows the accuracy is about 20%, the accuracy in 800 data rows is about 30% and 1200 data rows the accuracy is about 40%. The difficulty in using NMF is that we are not able to estimate the best dimension to reduce. In computer science, NMF is categorized as an NP-hard problem. Non-deterministic polynomial (NP) is defined as the solution can be guessed and verified in polynomial time; non-deterministic means that no particular rule is followed to make the guess. In this case, we are using the previous section's estimated dimension in the NFM model. In this model, the approximated matrix W and matrix H are not always the same. To ensure that the matrices are the same when running the code for the second time, we need to set parameter, "random_state" to be the same in the form of integers.

In Appendix F, we look into another analysis that we can do using NMF. In previous case, the parameter "n_components" is the best dimension to approximate matrix W and matrix H. Now, in this part, we let "n_components" be the number of topics. The data set that we use consists of 40 topics, then we set "n_components" to 40. Next, we set parameter "init" be "random", "solver" be “cd”, “beta_loss” be “Frobenius” and “random_state” be 0 for all 400, 800 and 1200 data rows. Since the combinations of parameters for 400, 800 and 1200 are the same, thus we get the same W and H matrices for all three cases. The matrix W is the document-topic matrix whereas matrix H is the topic-words matrix. Matrix W is used to assign each topic to a title. Matrix H is used to identify the most used words in a particular topic. For example, we choose Topic 30 which is Sports-Recreation and the top 5 words from this topic are “last”, “season”, “game”, “hr” and “good”. These top 5 words are the common words related to Sports-Recreation.

4.5 Time comparison between Singular Value Decomposition (SVD) and Non-Negative Matrix Factorization (NMF)

In this section, we compare the time needed for the matrix factorizations to calculate the approximated matrix. This comparison is between SVD and NMF. The time result is tabulated in Table 4.8 and 4.9.

Table 4.8: Time needed to calculate approximated matrix using SVD.

Number of data row	Time usage using original matrix (second)	Time usage using sparse matrix (second)
400	0.85436	0.76629
800	4.06100	3.75967
1200	11.12663	10.21182

Table 4.9: Time needed to calculate approximated matrix using NMF.

Number of data row	Time usage using original matrix (second)	Time usage using sparse matrix (second)
400	481.57546	455.14909
800	1271.99165	1182.60352
1200	2842.28285	2655.21590

The code for getting the results of Table 4.8 and Table 4.9 can be viewed in Appendix G. In these results, we also use another type of sparse matrix. A sparse matrix is a matrix where most of its entries are zero. From the two methods, when we increase the number of data rows the time needed has increased as well. When sparse matrix is used we obtain the same pattern of results when the number of data row is increased. But, the time needed is less as compared to when using the original size of matrix. In Python, there are codes that help to compress the dimension of sparse matrix by discarding the zero entries. The computation speed increases when dealing with smaller size of input matrix.

CHAPTER 5

CONCLUSION

This report shows the application of Singular Value Decomposition (SVD) and Non-Negative Matrix Factorization (NMF) in Latent Semantic Analysis (LSI). LSI is an important algorithm for searching a document by using keywords as input to return the document that has the highest similarity with the keywords. From the results, we conclude that SVD performs better than NMF in LSI. This is mainly because SVD has a way to find a suitable dimension, k, to reduce the dimension of the matrix whereas NMF does not have such a way to find the value of k. In future, we plan to find a better method which have a better approximation of k. Besides, in this project, we use only 400, 800 and 1200 data rows. This can be extended to more than 1200 data rows. From the result in Section 4.3, we have accuracy 70% to 80% when comparing the result between LSI without matrix factorization and LSI with matrix factorization. The accuracy may increase or reduce when we increase the data rows. It is too early to assume that as we have only used $\frac{1200}{681288} = 0.00176137 \approx 0\%$ of the whole data. Furthermore, sparse matrices shall be used to reduce the dimension of the matrix and more data rows shall be included to have a better result. There are some other methods that we have yet to explore that may help to improve the accuracy and time usage of the algorithm.

REFERENCES

Brownlee, J., 2018. *A Gentle Introduction to Matrix Factorization for Machine Learning*. [online] (09 August 2019) Available at: <https://machinelearningmastery.com/introduction-to-matrix-decompositions-for-machine-learning/> [Accessed 08 February 2021].

Brownlee, J., 2018. *How to Calculate the SVD from Scratch with Python*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/singular-value-decomposition-for-machine-learning/> [Accessed 15 June 2021].

Cergani, E., and Miettinen, P., 2013. *Discovering Relations Using Matrix Factorization Methods*. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management. San Francisco, California, US. New York, NY, USA: Association for Computing Machinery. Available at: <https://people.mpi-inf.mpg.de/ pmiettin/papers/cergani13discovering.pdf> [Accessed 01 July 2021].

Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K. and Harshman, R., 1990. Indexing by Latent Semantic Analysis. *Journal of the American society for information science*, 41(6), pp.391-407. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.1152&rep=rep1&type=pdf> [Accessed 14 February 2021].

Ensari, T., Chorowski, J. and Zurada, J.M., 2012. *Occluded face recognition using correntropy-based nonnegative matrix factorization*. In: 2012 11th International Conference on Machine Learning and Applications (Vol. 1, pp. 606-609). Boca Raton, Florida, USA, 12-15 December 2012. New York: IEEE. Available through: Universiti Tunku Abdul Rahman Library website <https://library.utar.edu.my/> [Accessed 26 February 2021].

Horn, R.A. and Johnson, C.R., 2013. *Matrix Analysis*. 2nd ed.

Kleinberg, J.M., 1999. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, [e-journal] 46(5), pp. 604-632. Available at: < http://www.cse.msu.edu/c̃se960/Papers/LinkAnalysis/auth.pdf> [Accessed 15 February 2021].

Ng, W.S. and Tan, W.W., 2021. Some properties of various types of matrix factorization. *ITM Web of Conferences*, [online] 36(03003), pp.9. https://doi.org/10.1051/itmconf/20213603003 [Accessed 5 February 2021].

Pandey, J.P. and Umrao, L.S., 2019. *Digital Image Processing using Singular Value Decomposition*. In: 2nd International Conference on Advanced Computing and Software Engineering (ICACSE) 2019. Sultanpur, India, 8-9 February 2019. Available through: SSRN website <https://www.ssrn.com/> [Accessed 14 February 2021].

Scikit-learn.org, 2009. *sklearn.decomposition.NMF — scikit-learn 0.21.3 documentation*. [online] Available at: <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html.> [Accessed 15 June 2021].

Stewart, G. W., 1993. On the Early History of the Singular Value Decomposition. *SIAM Review*, 35(4), pp. 551-556. Available through: Universiti Tunku Abdul Rahman website <https://library.utar.edu.my/> [Accessed 17 February 2021].

Tatman, R., 2017. *Blog Authorship Corpus*. Natural Language Processing data set. Kaggle. Available at: <https://www.kaggle.com/rtatman/blog-authorship-corpus> [Accessed 17 June 2021].

Vasireddy, J.L., 2009. *Applications of Linear Algebra to Information Retrieval*. Postgraduate. Georgia State University. Available at: <https://scholarworks.gsu.edu/cgi/viewcontent.cgi?article=1070&context=math_theses> [Accessed 14 February 2021].

Zurada, J.M., Ensari, T., Asl, E.H. and Chorowski, J., 2013. *Nonnegative matrix factorization and its application to pattern analysis and text mining*. In: 2013 Federated Conference on Computer Science and Information Systems, pp. 11-16. Kraków, Poland, 8-11 September 2013. Available through: IEEE Xplore website <https://ieeexplore.ieee.org/>. [Accessed 16 February 2021].

APPENDICES

APPENDIX A: Customize and Clean DataFrame

```
In [ ]: # Customize and Clean DataFrame
        def Customize_Cleaning_DataFrame(number_of_data_from_each_categoty):

            # find each of the category name and find the number of category
            topic = df['topic'].value_counts().index
            number_of_topic = df['topic'].value_counts().count()

            # extract to make a new DataFrame with 40x rows
            topic_category = topic.to_list()
            x = number_of_data_from_each_categoty
            df_0 = df[df['topic'] == topic_category[0]].head(x)
            df_1 = df[df['topic'] == topic_category[1]].head(x)
            df_2 = df[df['topic'] == topic_category[2]].head(x)
            df_3 = df[df['topic'] == topic_category[3]].head(x)
            df_4 = df[df['topic'] == topic_category[4]].head(x)
            df_5 = df[df['topic'] == topic_category[5]].head(x)
            df_6 = df[df['topic'] == topic_category[6]].head(x)
            df_7 = df[df['topic'] == topic_category[7]].head(x)
            df_8 = df[df['topic'] == topic_category[8]].head(x)
            df_9 = df[df['topic'] == topic_category[9]].head(x)
            df_10 = df[df['topic'] == topic_category[10]].head(x)
            df_11 = df[df['topic'] == topic_category[11]].head(x)
            df_12 = df[df['topic'] == topic_category[12]].head(x)
            df_13 = df[df['topic'] == topic_category[13]].head(x)
            df_14 = df[df['topic'] == topic_category[14]].head(x)
            df_15 = df[df['topic'] == topic_category[15]].head(x)
            df_16 = df[df['topic'] == topic_category[16]].head(x)
            df_17 = df[df['topic'] == topic_category[17]].head(x)
            df_18 = df[df['topic'] == topic_category[18]].head(x)
            df_19 = df[df['topic'] == topic_category[19]].head(x)
            df_20 = df[df['topic'] == topic_category[20]].head(x)
            df_21 = df[df['topic'] == topic_category[21]].head(x)
            df_22 = df[df['topic'] == topic_category[22]].head(x)
            df_23 = df[df['topic'] == topic_category[23]].head(x)
            df_24 = df[df['topic'] == topic_category[24]].head(x)
            df_25 = df[df['topic'] == topic_category[25]].head(x)
            df_26 = df[df['topic'] == topic_category[26]].head(x)
            df_27 = df[df['topic'] == topic_category[27]].head(x)
            df_28 = df[df['topic'] == topic_category[28]].head(x)
            df_29 = df[df['topic'] == topic_category[29]].head(x)
            df_30 = df[df['topic'] == topic_category[30]].head(x)
            df_31 = df[df['topic'] == topic_category[31]].head(x)
            df_32 = df[df['topic'] == topic_category[32]].head(x)
            df_33 = df[df['topic'] == topic_category[33]].head(x)
            df_34 = df[df['topic'] == topic_category[34]].head(x)
            df_35 = df[df['topic'] == topic_category[35]].head(x)
            df_36 = df[df['topic'] == topic_category[36]].head(x)
            df_37 = df[df['topic'] == topic_category[37]].head(x)
            df_38 = df[df['topic'] == topic_category[38]].head(x)
            df_39 = df[df['topic'] == topic_category[39]].head(x)

            # New DataFrame
            frames = [df_0, df_1, df_2, df_3, df_4, df_5, df_6, df_7, df_8, df_9,
                  df_10, df_11, df_12, df_13, df_14, df_15, df_16, df_17, df_18, df_19
        ,
                  df_20, df_21, df_22, df_23, df_24, df_25, df_26, df_27, df_28, df_29
        ,
                  df_30, df_31, df_32, df_33, df_34, df_35, df_36, df_37, df_38, df_39
        ]

            df_new = pd.concat(frames)

            #  text Processsing
            data_original = df_new

            # generating the index name
            index_number = len(data_original.index)
            number_of_document = [y + 1 for y in range(index_number)]
            number_of_document_string = [str(int) for int in number_of_document]
            index_name = ['Document ' + w for w in number_of_document_string]
            data_original.index = index_name

            # combine column topic and text
```

```
    data_original['blog'] = data_original['topic'] + data_original['text']

    # remove column
    data_original_remove = data_original.drop(['id',
                                               'gender',
                                               'age',
                                               'sign',
                                               'date',
                                               'text',
                                               'topic'], axis = 1)
    # generating the index name
    index_number = len(data_original_remove.index)
    number_of_document = [y + 1 for y in range(index_number)]
    number_of_document_string = [str(int) for int in number_of_document]
    index_name = ['Document ' + w for w in number_of_document_string]
    data_original_remove.index = index_name

    # extract feature
    feature = data_original_remove.iloc[:,0]

    # Data Preprocessing
    def nlp_process(processed_feature):
        # symtom comes from RegEX study
        for sentence in range(0, len(feature)):
            # remove all the special characters
            processed = re.sub(r'\W', ' ', str(feature[sentence]))
            # converting to lowercase
            processed = processed.lower()
            # remove digits
            processed = re.sub('\d+', ' ', processed)
            # remove all single characters
            processed = re.sub(r'\s+[a-zA-Z]\s+', ' ', processed)
            # remove single characters start from 1st characters
            processed = re.sub(r'\^[a-zA-Z]\s+', ' ', processed)
            # substituting multiple spaces with single space
            processed = re.sub(r'\s+', ' ', processed, flags = re.I)
            # removing prefixed 'b'
            processed = re.sub(r'^b\s+', '', processed)
            # remove symbols
            processed = re.sub(r'[^\w]', ' ', processed)
            # remove dot
            processed = re.sub(r'\.(?!\d)', '', processed)
            processed = re.sub(' +', ' ', processed)
            # remove extra space
            processed = re.sub('\s+', ' ', processed.strip())
            # remove all the special characters
            processed = re.sub(r'\W', ' ', processed)
            # append to list
            processed_feature.append(processed)
        return processed_feature
    review_list = []
    nlp_process(review_list)

    # change list to dataframe
    data_original_remove_preprocessing = pd.DataFrame(review_list, columns = [
'text'])
    # print(data_original_remove_preprocessing)
    # print()

    # remove stopwords
    data_original_remove_preprocessing['text'] = data_original_remove_preproce
ssing['text'].apply(lambda x: ' '.join([word for word in x.split() if word not
in (stop)]))

    # stemming reduce words back to its base form
    # Use English stemmer
    # stemmer = SnowballStemer("english")
    # data_clean['text'] = data_clean['text'].apply(lambda x: [stemmer.stem(y)
for y in x])
    # data_clean
    # for accuracy i will use lemmatization

    # lemmatization
    en_core = spacy.load('en_core_web_sm')
```

```
    data_original_remove_preprocessing['lemmatized'] = data_original_remove_pr
eprocessing['text'].apply(lambda x: " ".join([y.lemma_ for y in en_core(x)]))

    # remove accents
    def remove_accents(input_str):
        nfkd_form = unicodedata.normalize('NFKD', input_str)
        only_ascii = nfkd_form.encode('ASCII', 'ignore')
        return only_ascii
    data_original_remove_preprocessing['lemmatized_remove_accents'] = data_ori
ginal_remove_preprocessing['lemmatized'].apply(remove_accents)

    # change bytes back to stringavailable
    #str(data_clean['lemmatized'], 'UTF-8')
    data_original_remove_preprocessing['lemmatized_remove_accents'] = data_ori
ginal_remove_preprocessing['lemmatized_remove_accents'].str.decode("utf-8")

    # only need the column that is clean
    data_clean = data_original_remove_preprocessing.drop(['text', 'lemmatized'
], axis = 1)

    # rename dataframe row
    # calculate the number of row in the dataframe
    row_number = len(data_clean.index)
    # put the number of rows into list
    row_number_list = [y + 1 for y in range(row_number)]
    # change the type list to string
    row_number_list_string = [str(int) for int in row_number_list]
    # put the rows name into the list
    row_name = ['Document ' + w for w in row_number_list_string]
    # change the dataframe row name
    data_clean.index = row_name
    return data_clean, data_original
```

In [ ]:
```
# data frame 400 rows of data
df_400, df_original_400 = Customize_Cleaning_DataFrame(10)
df_400.head(20)
```

In [ ]:
```
# DateFrame 800 rows of data
df_800, df_original_800 = Customize_Cleaning_DataFrame(20)
df_800.head(20)
```

In [ ]:
```
# data frame 1200 rows of data
df_1200, df_original_1200 = Customize_Cleaning_DataFrame(30)
df_1200.head(20)
```

APPENDIX B: Convert the Dataframe

In [ ]:
```
def Convert_DataFrame(df):
    # dataframe to list
    text_original = df.values.tolist()
    # un-nest each list stored in the list of list
    sublist_original = ['lemmatized_remove_accents']
    text_original = [val for sublist_original in text_original for val in subl
ist_original]
    # BOW model for text original
    # fit the tokenizer on text
    model_original.fit_on_texts(text_original)
    # get the bag of words representation
    rep = model_original.texts_to_matrix(text_original, mode = 'count')
    # transpose the matrix rep
    rep_transpose = rep.transpose()
    # convert array to dataframe
    df_original = pd.DataFrame(rep_transpose)
    # droppping the first row
    # first row does not hold any important information
    df_original = df_original.drop(0, axis = 0)
    # exract the key dictionary into a list
    key_original = list(model_original.word_index.keys())
    # change index to key
    df_original.index = key_original
    # calculate the number of column in the dataframe
    column_number_original = len(df_original.columns)
    # put the number of columns into a list
    number_of_document_original = [y + 1 for y in range(column_number_original
)]
    # value in the list change to string
    number_of_document_string_original = [str(int) for int in number_of_docume
nt_original]
    # put column name into the list
    column_name_original = ['Document ' + w for w in number_of_document_string
_original]
    # change the dataframe column name
    df_original.columns = column_name_original
    # change value to int
    df_original = df_original.astype(int)
    return df_original, key_original
```

In [ ]:
```
# words - document dataframe of 400
df_400_WD, key_400 = Convert_DataFrame(df_400)
df_400_WD
```

In [ ]:
```
# words - document dataframe of 400
df_800_WD, key_800 = Convert_DataFrame(df_800)
df_800_WD
```

In [ ]:
```
# words - document dataframe of 1200
df_1200_WD, key_1200 = Convert_DataFrame(df_1200)
df_1200_WD
```

APPENDIX C: Change user input to word-document DataFrame

In [ ]:
```
# data preparation of user data
# get user input
# input: Food and Travel
# x = str(input('Enter the blog: '))
x = ['Food and Travel']
text_input = [x]
text_input = pd.DataFrame(text_input, columns = ['Text'])

# get feature
feature = text_input.iloc[:,0]

# text preprocessing
def nlp_process(processed_feature):
    # symtom comes from RegEX study
    for sentence in range(0, len(feature)):
        # remove all the special characters
        processed = re.sub(r'\W', ' ', str(feature[sentence]))
        # converting to lowercase
        processed = processed.lower()
        # remove digits
        processed = re.sub('\d+', ' ', processed)
        # remove all single characters
        processed = re.sub(r'\s+[a-zA-Z]\s+', ' ', processed)
        # remove single characters start from 1st characters
        processed = re.sub(r'\^[a-zA-Z]\s+', ' ', processed)
        # substituting multiple spaces with single space
        processed = re.sub(r'\s+', ' ', processed, flags = re.I)
        # removing prefixed 'b'
        processed = re.sub(r'^b\s+', '', processed)
        # remove symbols
        processed = re.sub(r'[^\w]', ' ', processed)
        # remove dot
        processed = re.sub(r'\.(?!\d)', '', processed)
        processed = re.sub(' +', ' ', processed)
        # remove extra space
        processed = re.sub('\s+', ' ', processed.strip())
        # remove all the special characters
        processed = re.sub(r'\W', ' ', processed)
        # append to list
        processed_feature.append(processed)
    return processed_feature
review_list = []
nlp_process(review_list)

# change list into dataframe
data_user_input = pd.DataFrame(review_list, columns = ['Text'])

# remove stop words
stop = stopwords.words('english')
data_user_input['Text'] = data_user_input['Text'].apply(lambda x: ' '.join(wor
d for word in x.split() if word not in (stop)))

# lemmatization
en_core = spacy.load('en_core_web_sm')
data_user_input['Text'] = data_user_input['Text'].apply(lambda x: ' '.join(y.l
emma_ for y in en_core(x)))

text_user = data_user_input.values.tolist()

# un-nest each list stored in the list of list
sublist_input = ['Text']
text_user = [val for sublist_input in text_user for val in sublist_input]

# fit the tokenizer on text
model_user.fit_on_texts(text_user)
# get the bag of words representation
rep = model_user.texts_to_matrix(text_user, mode = 'count')
# transpose the matrix rep
rep_transpose = rep.transpose()
# convert array to dataframe
df_user = pd.DataFrame(rep_transpose)
# dropping the first row
# first row do not hold any important information
```

```
df_user = df_user.drop(0, axis = 0)
# extract the key dictionary into a list
key_user = list(model_user.word_index.keys())
# change index to key
df_user.index = key_user
# calculate the numbrt of column in the dataframe
column_number = len(df_user.columns)
# put the number of columms into a list
number_of_document = [y + 1 for y in range(column_number)]
# value in the list change to string
number_of_document_string = [str(int) for int in number_of_document]
# put the column name into the list
column_name = ['Document ' + w for w in number_of_document_string]
# change the dataframe column name
df_user.columns = column_name
# change value to int
df_user = df_user.astype(int)
df_user
```

In [ ]:
```
def Convert_user_DataFrame(key):
    x = len(key)
    list = []
    for a in range(x):
        list.append(0)
    # create a new dataframe which have the same index as the original datafra
me
    df_user_2 = pd.DataFrame(list)
    df_user_2.index = key
    df_user_transpose = df_user.transpose()
    df_user_2_transpose = df_user_2.transpose()
    name = ['Document 1']
    df_user_2_transpose.index = name
    df_user_2_transpose.update(df_user_transpose)
    df_user_2 = df_user_2_transpose.transpose()
    return df_user_2
```

In [ ]:
```
# DataFrame user for 400
df_user_400 = Convert_user_DataFrame(key_400)
df_user_400
```

In [ ]:
```
# DataFrame user for 800
df_user_800 = Convert_user_DataFrame(key_800)
df_user_800
```

In [ ]:
```
# DataFrame user for 1200
df_user_1200 = Convert_user_DataFrame(key_1200)
df_user_1200
```

APPENDIX D: LSI without Matrix Factorization

```
In [ ]: # function to compute matrix multiplication
        def matrix_multiplication(original_text, user_text):
            # convert dataframe into array for matrix multiplication
            original_text_matrix = original_text.values
            user_text_matrix = user_text.values
            # transpose the matrix original_text_matrix
            original_text_matrix = original_text_matrix.transpose()
            # matrix multiplication
            matrix_mul = np.dot(original_text_matrix, user_text_matrix)
            # generating the index name
            column_number = len(original_text.columns)
            number_of_document = [y + 1 for y in range(column_number)]
            number_of_document_string = [str(int) for int in number_of_document]
            column_name = ['Document ' + w for w in number_of_document_string]
            # convert matrix into a dataframe
            data_final = pd.DataFrame(matrix_mul, index = column_name)
            return data_final
```

```
In [ ]: # function to get LSI result without matrix factorization
        def LSI(df, df_user, df_original):
            # result without using matrix factorization
            result_lsi = matrix_multiplication(df, df_user)
            # find the top 10 largest value document
            result_lsi_index = result_lsi[0].nlargest(10).index
            # transpose the original dataframe
            df_original_transpose = df_original.transpose()
            # return the search of the user
            final_result_lsi = df_original_transpose[result_lsi_index]
            # transpose
            final_result_lsi = final_result_lsi.transpose()

            result = result_lsi[0].nlargest(10)
            return result
```

```
In [ ]: # result 400 data
        result_lsi_400 = LSI(df_400_WD, df_user_400, df_original_400)
        result_lsi_400
```

```
In [ ]: # result 800 data
        result_lsi_800 = LSI(df_800_WD, df_user_800, df_original_800)
        result_lsi_800
```

```
In [ ]: # result 1200 data
        result_lsi_1200 = LSI(df_1200_WD, df_user_1200, df_original_1200)
        result_lsi_1200
```

APPENDIX E: Applying SVD in LSI

In [ ]:
```
# getting the sigma value
def LSI_SVD(df):
    # convert dataframe to array
    arr_data_original = df.to_numpy()
    # singular value decomposition
    U, s, VT = linalg.svd(arr_data_original, full_matrices = False)
    # reconstruct Sigma matrix
    Sigma = np.zeros((arr_data_original.shape[0], arr_data_original.shape[1]))
    # populate Sigma with n x n diagonal matrix
    Sigma[:arr_data_original.shape[1], :arr_data_original.shape[1]] = np.diag(
s)
    return s, U, Sigma, VT
```

In [ ]:
```
Sigma_400, Matrix_U_400, Matrix_Sigma_400, Matrix_VT_400 = LSI_SVD(df_400_WD)
Sigma_800, Matrix_U_800, Matrix_Sigma_800, Matrix_VT_800 = LSI_SVD(df_800_WD)
Sigma_1200, Matrix_U_1200, Matrix_Sigma_1200, Matrix_VT_1200 = LSI_SVD(df_1200
_WD)
```

In [ ]:
```
# function to sum up the sigma
def sum_sigma(k, sigma):
    sum_value = 0
    for n in range(k):
        sum_value = sum_value + sigma[n]**2
    return sum_value
```

In [ ]:
```
# function to calculate the k needed
def SVD_power(power ,sigma ,df_original):
    power_level = power
    sum_total_sigma = sum_sigma(len(df_original), sigma)
    ratio = 0
    k = 0

    while ratio <= power:
        k_previous = k
        ratio_previous = ratio
        k = k + 1
        k_sigma_sum = sum_sigma(k, sigma)
        ratio = k_sigma_sum / sum_total_sigma
    else:
        return k_previous
```

In [ ]:
```
# to get the value of k for dimensional reduction

k_400_85_percent = SVD_power(0.85 ,Sigma_400 ,df_original_400)
k_400_90_percent = SVD_power(0.90 ,Sigma_400 ,df_original_400)
k_400_95_percent = SVD_power(0.95 ,Sigma_400 ,df_original_400)

k_800_85_percent = SVD_power(0.85 ,Sigma_800 ,df_original_800)
k_800_90_percent = SVD_power(0.90 ,Sigma_800 ,df_original_800)
k_800_95_percent = SVD_power(0.95 ,Sigma_800 ,df_original_800)

k_1200_85_percent = SVD_power(0.85 ,Sigma_1200 ,df_original_1200)
k_1200_90_percent = SVD_power(0.90 ,Sigma_1200 ,df_original_1200)
k_1200_95_percent = SVD_power(0.95 ,Sigma_1200 ,df_original_1200)
```

In [ ]:
```
# function to get result from the approximate reduction matrix

def result_approximate_matrix(k, U, Sigma, VT, df_original, df_WD, df_user):
    U_approx = U[:,:k]
    Sigma_approx = Sigma[:k,:k]
    VT_approx = VT[:k,:]
    matrix_svd_approx = U_approx.dot(Sigma_approx.dot(VT_approx))

    # convert numpy to dataframe
    df_approx = pd.DataFrame(matrix_svd_approx)
    # change name for index
    df_approx.index = df_WD.index
    # change name for columns
    column_number_approx = len(df_approx.columns)
    number_of_document_approx = [y + 1 for y in range(column_number_approx)]
    number_of_document_string_approx = [str(int) for int in number_of_document
_approx]
    column_name_approx = ['Document ' + w for w in number_of_document_string_a
pprox]
    df_approx.columns = column_name_approx

    # LSI with matrix factorization
    result_lsi_Matrix_Factorization = matrix_multiplication(df_approx, df_user
)

    # find the top 10 largest value document
    result_lsi_Matrix_Factorization_index = result_lsi_Matrix_Factorization[0]
.nlargest(10).index

    # transpose the original dataframe
    df_original_transpose = df_original.transpose()

    # return the search of the user
    final_result_lsi_Matrix_Factorization = df_original_transpose[result_lsi_M
atrix_Factorization_index]

    # transpose
    final_result_lsi_Matrix_Factorization = final_result_lsi_Matrix_Factorizat
ion.transpose()

    result = result_lsi_Matrix_Factorization[0].nlargest(10)

    return result
```

In [ ]:
```
# result for 400 rows data
result_85_percent_400 = result_approximate_matrix(k_400_85_percent, Matrix_U_4
00, Matrix_Sigma_400, Matrix_VT_400, df_original_400, df_400_WD, df_user_400)
result_90_percent_400 = result_approximate_matrix(k_400_90_percent, Matrix_U_4
00, Matrix_Sigma_400, Matrix_VT_400, df_original_400, df_400_WD, df_user_400)
result_95_percent_400 = result_approximate_matrix(k_400_95_percent, Matrix_U_4
00, Matrix_Sigma_400, Matrix_VT_400, df_original_400, df_400_WD, df_user_400)

print(result_85_percent_400, '\n')
print(result_90_percent_400, '\n')
print(result_95_percent_400, '\n')
```

In [ ]:
```
# result for 800 rows data
result_85_percent_800 = result_approximate_matrix(k_800_85_percent, Matrix_U_8
00, Matrix_Sigma_800, Matrix_VT_800, df_original_800, df_800_WD, df_user_800)
result_90_percent_800 = result_approximate_matrix(k_800_90_percent, Matrix_U_8
00, Matrix_Sigma_800, Matrix_VT_800, df_original_800, df_800_WD, df_user_800)
result_95_percent_800 = result_approximate_matrix(k_800_95_percent, Matrix_U_8
00, Matrix_Sigma_800, Matrix_VT_800, df_original_800, df_800_WD, df_user_800)

print(result_85_percent_800, '\n')
print(result_90_percent_800, '\n')
print(result_95_percent_800, '\n')
```

```
In [ ]: # result for 1200 rows data
        result_85_percent_1200 = result_approximate_matrix(k_1200_85_percent, Matrix_U
        _1200, Matrix_Sigma_1200, Matrix_VT_1200, df_original_1200, df_1200_WD, df_use
        r_1200)
        result_90_percent_1200 = result_approximate_matrix(k_1200_90_percent, Matrix_U
        _1200, Matrix_Sigma_1200, Matrix_VT_1200, df_original_1200, df_1200_WD, df_use
        r_1200)
        result_95_percent_1200 = result_approximate_matrix(k_1200_95_percent, Matrix_U
        _1200, Matrix_Sigma_1200, Matrix_VT_1200, df_original_1200, df_1200_WD, df_use
        r_1200)

        print(result_85_percent_1200, '\n')
        print(result_90_percent_1200, '\n')
        print(result_95_percent_1200, '\n')
```

```
In [ ]: # function to get the x-axis
        def x_axis(df_original):
            x_axis = []
            for n in range(len(df_original.index)):
                x_axis.append(n + 1)
            return x_axis
```

```
In [ ]: # getting the corresponsding x_axis
        x_axis_400 = x_axis(df_original_400)
        x_axis_800 = x_axis(df_original_800)
        x_axis_1200 = x_axis(df_original_1200)
```

```
In [ ]: # Frobenius norm with 8 decimal point
        def FN_decimal(df_original, df_WD, U, Sigma, VT):
            y_axis_decimal = []
            for b in range(len(df_original.index)):
                # dimensional reduction
                n = b + 1
                U_reduce_loop = U[:,:n]
                Sigma_reduce_loop = Sigma[:n,:n]
                VT_reduce_loop = VT[:n,:]
                matrix_svd_reduce_loop = U_reduce_loop.dot(Sigma_reduce_loop.dot(VT_re
        duce_loop))
                matrix_svd_reduce_loop_round = np.around(matrix_svd_reduce_loop, 8)

                df_WD_matrix = df_WD.to_numpy()
                matrix_diff_loop = df_WD_matrix - matrix_svd_reduce_loop_round

                # Frobenius norm
                Fron_norm_loop = norm(matrix_diff_loop, ord = 'fro')

                # insert into the array
                y_axis_decimal.append(Fron_norm_loop)

            return y_axis_decimal
```

```
In [ ]: y_axis_decimal_400 = FN_decimal(df_original_400, df_400_WD, Matrix_U_400, Matr
        ix_Sigma_400, Matrix_VT_400)
        y_axis_decimal_800 = FN_decimal(df_original_800, df_800_WD, Matrix_U_800, Matr
        ix_Sigma_800, Matrix_VT_800)
        y_axis_decimal_1200 = FN_decimal(df_original_1200, df_1200_WD, Matrix_U_1200,
        Matrix_Sigma_1200, Matrix_VT_1200)
```

In [ ]:
```
# Frobenius norm with integer
def FN_integer(df_original, df_WD, U, Sigma, VT):
    y_axis_integer = []
    for b in range(len(df_original.index)):
        # dimensional redunction
        n = b + 1
        if n > len(df_original.index):
            break
        U_reduce_loop = U[:,:n]
        Sigma_reduce_loop = Sigma[:n,:n]
        VT_reduce_loop = VT[:n,:]
        matrix_svd_reduce_loop = U_reduce_loop.dot(Sigma_reduce_loop.dot(VT_re
duce_loop))
        matrix_svd_reduce_loop_round = np.around(matrix_svd_reduce_loop)

        df_WD_matrix = df_WD.to_numpy()
        matrix_diff_loop = df_WD_matrix - matrix_svd_reduce_loop_round

        # Frobenius norm
        Fron_norm_loop = norm(matrix_diff_loop, ord = 'fro')

        # insert into the array
        y_axis_integer.append(Fron_norm_loop)

    return y_axis_integer
```

In [ ]:
```
y_axis_integer_400 = FN_integer(df_original_400, df_400_WD, Matrix_U_400, Matr
ix_Sigma_400, Matrix_VT_400)
y_axis_integer_800 = FN_integer(df_original_800, df_800_WD, Matrix_U_800, Matr
ix_Sigma_800, Matrix_VT_800)
y_axis_integer_1200 = FN_integer(df_original_1200, df_1200_WD, Matrix_U_1200,
Matrix_Sigma_1200, Matrix_VT_1200)
```

In [ ]:
```
# function to plot graph
def graph(x_axis, y_axis_decimal, y_axis_integer, s, t):
    fig, ax = plt.subplots(figsize = (14, 8))
    x = x_axis
    y_decimal = y_axis_decimal
    y_integer = y_axis_integer
    ax.plot(x,y_decimal, label = 'decimal', color = 'b')
    ax.plot(x,y_integer, label = 'integer', color = 'r')
    plt.legend
    # label x-axis
    plt.xlabel('Number of n')
    # label y_axis
    plt.ylabel('Frobenious Norm')
    plot = plt.show()

    line2d_1 = plt.plot(x,y_decimal)
    xvalues_decimal = line2d_1[0].get_xdata()
    yvalues_decimal = line2d_1[0].get_ydata()

    line2d_2 = plt.plot(x,y_integer)
    xvalues_integer = line2d_2[0].get_xdata()
    yvalues_integer = line2d_2[0].get_ydata()

    df_plot_graph = pd.DataFrame([xvalues_decimal, yvalues_decimal, yvalues_in
teger])
    df_plot_graph = df_plot_graph.transpose()
    df_plot_graph.columns = ['Number of k', 'decimal', 'integer']
    df_plot_graph.index = [a + 1 for a in range(len(df_plot_graph.index))]
    df_plot_graph = df_plot_graph.iloc[s:t, :]

    return plot, df_plot_graph
```

In [ ]:
```
# 400 data set
plot_400, df_plot_graph_400 = graph(x_axis_400, y_axis_decimal_400, y_axis_int
eger_400, 367, 375)
print(plot_400, '\n')
print(df_plot_graph_400)
```

```
In [ ]: # 800 data set
        plot_800, df_plot_graph_800 = graph(x_axis_800, y_axis_decimal_800, y_axis_int
        eger_800, 695, 705)
        print(plot_800, '\n')
        print(df_plot_graph_800)
```

```
In [ ]: # 1200 data set
        plot_1200, df_plot_graph_1200 = graph(x_axis_1200, y_axis_decimal_1200, y_axis
        _integer_1200, 1025, 1035)
        print(plot_1200, '\n')
        print(df_plot_graph_1200)
```

```
In [ ]: # getting the value of k using Frobenius norm
        def Frobenius_norm(df_WD, U, Sigma, VT):
            Frob_norm_now = 0
            Frob_norm_previous = 1
            n = 0

            while Frob_norm_now != Frob_norm_previous:
                Frob_norm_previous = Frob_norm_now
                n = n + 1
                if n > len(df_WD.columns):
                    break
                U_loop = U[:,:n]
                Sigma_loop = Sigma[:n,:n]
                VT_loop = VT[:n,:]
                matrix_svd_loop = U_loop @ Sigma_loop @ VT_loop
                matrix_svd_loop_round = np.around(matrix_svd_loop)

                df_WD_matrix = df_WD.to_numpy()
                matrix_diff_loop_2 = df_WD_matrix - matrix_svd_loop_round

                # Frobenius norm
                Frob_diff_loop_2 = norm(matrix_diff_loop_2, ord = 'fro')
                Frob_diff_loop_2 = round(Frob_diff_loop_2, 10)
                Frob_norm_now = Frob_diff_loop_2
            else:
                return n - 1
```

```
In [ ]: # choose the parameter k
        k_400 = Frobenius_norm(df_400_WD, Matrix_U_400, Matrix_Sigma_400, Matrix_VT_40
        0)
        k_800 = Frobenius_norm(df_800_WD, Matrix_U_800, Matrix_Sigma_800, Matrix_VT_80
        0)
        k_1200 = Frobenius_norm(df_1200_WD, Matrix_U_1200, Matrix_Sigma_1200, Matrix_V
        T_1200)

        print(k_400, '\n')
        print(k_800, '\n')
        print(k_1200, '\n')
```

```
In [ ]: # final result of the LSI using Frobenius norm as the objective function 400 d
        ata set
        LSI_result_400 = result_approximate_matrix(k_400, Matrix_U_400, Matrix_Sigma_4
        00, Matrix_VT_400, df_original_400, df_400_WD, df_user_400)
        LSI_result_400
```

```
In [ ]: # final result of the LSI using Frobenius norm as the objective function 800 d
        ata set
        LSI_result_800 = result_approximate_matrix(k_800, Matrix_U_800, Matrix_Sigma_8
        00, Matrix_VT_800, df_original_800, df_800_WD, df_user_800)
        LSI_result_800
```

```
In [ ]: # final result of the LSI using Frobenius norm as the objective function 400 d
        ata set
        LSI_result_1200 = result_approximate_matrix(k_1200, Matrix_U_1200, Matrix_Sigm
        a_1200, Matrix_VT_1200, df_original_1200, df_1200_WD, df_user_1200)
        LSI_result_1200
```

APPENDIX F: Applying NMF in LSI

In [ ]:
```
def parameter_tuning(n, df_WD, init_para, solver_para, beta_loss_para):
    df_WD_Matrix = df_WD.to_numpy()
    df_WD_Sparse_Matrix = csr_matrix(df_WD_Matrix)
    nmf_model = NMF(n_components = n, init = init_para, solver = solver_para,
beta_loss = beta_loss_para, random_state = 0)
    W = nmf_model.fit_transform(df_WD_Matrix)
    H = nmf_model.components_

    Approx_matrix = np.dot(W,H)
    diff_matrix = df_WD_Matrix - Approx_matrix

    # frobenius norm
    Frob_norm = norm(diff_matrix, ord = 'fro')
    return Frob_norm
```

In [ ]:
```
# 400 row data
FN_400_1 = parameter_tuning(370, df_400_WD, 'random', 'cd', 'frobenius')
FN_400_2 = parameter_tuning(370, df_400_WD, 'random', 'mu', 'frobenius')
FN_400_3 = parameter_tuning(370, df_400_WD, 'random', 'mu', 'kullback-leibler'
)
FN_400_4 = parameter_tuning(370, df_400_WD, 'nndsvd', 'cd', 'frobenius')
FN_400_5 = parameter_tuning(370, df_400_WD, 'nndsvd', 'mu', 'frobenius')
FN_400_6 = parameter_tuning(370, df_400_WD, 'nndsvd', 'mu', 'kullback-leibler'
)


print(FN_400_1)
print(FN_400_2)
print(FN_400_3)
print(FN_400_4)
print(FN_400_5)
print(FN_400_6)
```

In [ ]:
```
# 800 row data
FN_800_1 = parameter_tuning(700, df_800_WD, 'random', 'cd', 'frobenius')
FN_800_2 = parameter_tuning(700, df_800_WD, 'random', 'mu', 'frobenius')
FN_800_3 = parameter_tuning(700, df_800_WD, 'random', 'mu', 'kullback-leibler'
)
FN_800_4 = parameter_tuning(700, df_800_WD, 'nndsvd', 'cd', 'frobenius')
FN_800_5 = parameter_tuning(700, df_800_WD, 'nndsvd', 'mu', 'frobenius')
FN_800_6 = parameter_tuning(700, df_800_WD, 'nndsvd', 'mu', 'kullback-leibler'
)

print(FN_800_1)
print(FN_800_2)
print(FN_800_3)
print(FN_800_4)
print(FN_800_5)
print(FN_800_6)
```

In [ ]:
```
# 1200 row data
FN_1200_1 = parameter_tuning(1032, df_1200_WD, 'random', 'cd', 'frobenius')
FN_1200_2 = parameter_tuning(1032, df_1200_WD, 'random', 'mu', 'frobenius')
FN_1200_3 = parameter_tuning(1032, df_1200_WD, 'random', 'mu', 'kullback-leibl
er')
FN_1200_4 = parameter_tuning(1032, df_1200_WD, 'nndsvd', 'cd', 'frobenius')
FN_1200_5 = parameter_tuning(1032, df_1200_WD, 'nndsvd', 'mu', 'frobenius')
FN_1200_6 = parameter_tuning(1032, df_1200_WD, 'nndsvd', 'mu', 'kullback-leibl
er')

print(FN_800_1)
print(FN_800_2)
print(FN_800_3)
print(FN_800_4)
print(FN_800_5)
print(FN_800_6)
```

In [ ]:
```
# finding the result
def result_approximate_matrix_NMF(n, df_original, df_WD, df_user, init_para, s
olver_para, beta_loss_para):
    # convert the dataframe into numpy array
    df_WD_matrix = df_WD.to_numpy()
    # convert matrix into sparse matrix
    df_WD_sparse_matrix = csr_matrix(df_WD_matrix)
    # setting the NMF model
    model = NMF(n_components = n, init = init_para, solver = solver_para, beta
_loss = beta_loss_para, random_state = 0)
    W = model.fit_transform(df_WD_sparse_matrix)
    H = model.components_
    # multiply matrix W and H
    matrix_NMF_approx = W @ H
    # convert numpy to dataframe
    df_approx = pd.DataFrame(matrix_NMF_approx)
    df_approx.index = df_WD.index
    # change name for columns
    column_number_approx = len(df_approx.columns)
    number_of_document_approx = [y + 1 for y in range(column_number_approx)]
    number_of_document_string_approx = [str(int) for int in number_of_document
_approx]
    column_name_approx = ['Document ' + w for w in number_of_document_string_a
pprox]
    df_approx.columns = column_name_approx

    # LSI with matrix factorization
    result_lsi_Matrix_Factorization = matrix_multiplication(df_approx, df_user
)

    # find the top 10 largest value document
    result_lsi_Matrix_Factorization_index = result_lsi_Matrix_Factorization[0]
.nlargest(10).index

    # transpose the original dataframe
    df_original_transpose = df_original.transpose()

    # return the search of the user
    final_result_lsi_Matrix_Factorization = df_original_transpose[result_lsi_M
atrix_Factorization_index]

    # transpose
    final_result_lsi_Matrix_Factorization = final_result_lsi_Matrix_Factorizat
ion.transpose()

    result = result_lsi_Matrix_Factorization[0].nlargest(10)

    return result
```

In [ ]:
```
# result for 400 data row
LSI_NMF_result_400 = result_approximate_matrix_NMF(370, df_original_400, df_40
0_WD, df_user_400, 'random', 'cd', 'frobenius')
LSI_NMF_result_400
```

In [ ]:
```
# result for 800 data row
LSI_NMF_result_800 = result_approximate_matrix_NMF(700, df_original_800, df_80
0_WD, df_user_800, 'nndsvd', 'cd', 'frobenius')
LSI_NMF_result_800
```

In [ ]:
```
# result for 1200 data row
LSI_NMF_result_1200 = result_approximate_matrix_NMF(1032, df_original_1200, df
_1200_WD, df_user_1200, 'nndsvd', 'cd', 'frobenius')
LSI_NMF_result_1200
```

```
In [ ]: df_WD_sparse_matrix = csr_matrix(df_400_WD.to_numpy().transpose())

        model = NMF(n_components = 40, init = 'random', solver = 'cd', beta_loss = 'fr
        obenius', random_state = 0)
        W = model.fit_transform(df_WD_sparse_matrix)
        H = model.components_
        print(W.shape,'\n')
        print(H.shape)
```

```
In [ ]: index_number = 40
        number_of_document = [y + 1 for y in range(index_number)]
        number_of_document_string = [str(int) for int in number_of_document]
        index_name = ['Topic ' + w for w in number_of_document_string]

        topic_word = pd.DataFrame(H.round(3))
        topic_word.index = index_name
        topic_word.columns = df_400_WD.index
        topic_word
```

```
In [ ]: topic = df['topic'].value_counts().index
        topic
```

```
In [ ]: index_name_2 = []
        for k in range(len(df_original_400)):
            index_name_2.append(df_original_400['topic'][k])

        index_name_2
```

```
In [ ]: topic_doc = pd.DataFrame(W.round(5))
        topic_doc.columns = index_name
        topic_doc.index = index_name_2
        topic_doc
```

```
In [ ]: topic_doc.reset_index().groupby('index').mean().idxmax()
```

```
In [ ]: topic_word.T.sort_values(by = 'Topic 1', ascending = False).head(5)
```

```
In [ ]: topic_word.T.sort_values(by = 'Topic 30', ascending = False).head(5)
```

APPENDIX G: Compare the time taken of SVD and NMF to find the approximate matrix

In [ ]:
```
# function to calculate svd
def SVD_calculation(dimension, df_WD):
    # start time
    start = time.time()
    # convert dataframe to numpy array
    df_WD_matrix = df_WD.to_numpy()
    # singular value decomposition
    U, s, VT = linalg.svd(df_WD_matrix, full_matrices = False)
    # reconstruct Sigma matrix
    Sigma = np.zeros((df_WD_matrix.shape[0], df_WD_matrix.shape[1]))
    # populate Sigma with n x n diagonal matrix
    Sigma[:df_WD_matrix.shape[1], :df_WD_matrix.shape[1]] = np.diag(s)
    # truncated SVD
    U_reduce = U[:,:dimension]
    Sigma_reduce = Sigma[:dimension, :dimension]
    VT_reduce = VT[:dimension,:]
    Approx_matrix = U_reduce @ Sigma_reduce @ VT_reduce
    # end time
    end = time.time()
    # time difference
    diff = np.around(end - start, 5)
    return diff
```

In [ ]:
```
# Time Taken
time_400_SVD = SVD_calculation(370, df_400_WD)
time_800_SVD = SVD_calculation(700, df_800_WD)
time_1200_SVD = SVD_calculation(1032, df_1200_WD)

print(time_400_SVD)
print(time_800_SVD)
print(time_1200_SVD)
```

In [ ]:
```
# function to calculate svd with sparse matrix
def SVD_Sparse_calculation(dimension, df_WD):
    # start time
    start = time.time()
    # convert dataframe to sparse matrix
    df_WD_sparse_matrix = csr_matrix(df_WD.to_numpy(), dtype = float)
    # singular value decomposition
    U, s, VT = svds(df_WD_sparse_matrix, k = dimension)
    # reconstruct Sigma
    Sigma = np.zeros((dimension, dimension))
    Sigma[:dimension, :dimension] = np.diag(s)
    Approx_matrix = U @ Sigma @ VT
    # end time
    end = time.time()
    # time difference
    diff = np.around(end - start, 5)
    return diff
```

In [ ]:
```
time_400_SVD_sparse = SVD_Sparse_calculation(370, df_400_WD)
time_800_SVD_sparse = SVD_Sparse_calculation(700, df_800_WD)
time_1200_SVD_sparse = SVD_Sparse_calculation(1032, df_1200_WD)

print(time_400_SVD_sparse)
print(time_800_SVD_sparse)
print(time_1200_SVD_sparse)
```

In [ ]:
```
# function to calculate NMF
def NMF_calculation(dimension, df_WD):
    # start time
    start = time.time()

    df_WD_matrix = df_WD.to_numpy()
    nmf_model = NMF(n_components = dimension, init = 'random', solver = 'cd',
beta_loss = 'frobenius', random_state = 0)
    W = nmf_model.fit_transform(df_WD_matrix)
    H = nmf_model.components_

    Approx_matrix = W @ H

    # end time
    end = time.time()

    # diff
    diff = np.around(end - start, 5)
    return diff
```

In [ ]:
```
# time taken
time_400_NMF = NMF_calculation(370, df_400_WD)
time_800_NMF = NMF_calculation(700, df_800_WD)
time_1200_NMF = NMF_calculation(1032, df_1200_WD)

print(time_400_NMF)
print(time_800_NMF)
print(time_1200_NMF)
```

In [ ]:
```
# function to calculate NMF
def NMF_Sparse_calculation(dimension, df_WD):
    # start time
    start = time.time()

    df_WD_sparse_matrix = csr_matrix(df_WD.to_numpy())
    nmf_model = NMF(n_components = dimension, init = 'random', solver = 'cd',
beta_loss = 'frobenius', random_state = 0)
    W = nmf_model.fit_transform(df_WD_sparse_matrix)
    H = nmf_model.components_

    Approx_matrix = W @ H

    # end time
    end = time.time()

    # diff
    diff = np.around(end - start, 5)
    return diff
```

In [ ]:
```
# time taken
time_400_NMF_sparse = NMF_Sparse_calculation(370, df_400_WD)
time_800_NMF_sparse = NMF_Sparse_calculation(700, df_800_WD)
time_1200_NMF_sparse = NMF_Sparse_calculation(1032, df_1200_WD)

print(time_400_NMF_sparse)
print(time_800_NMF_sparse)
print(time_1200_NMF_sparse)
```