ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 202E

NUMERICAL METHODS PROJECT 2 REPORT

CRN	:	21333
LECTURER	:	Enver Özdemir
PROJECT NO	:	2
STUDENT	:	Madina Alzhanova 150220939

SPRING 2024

Contents

1 INTRODUCTION

This project aims to implement Latent Semantic Analysis (LSI) by using Singular Value Decomposition (SVD). LSI is a method for indexing and retrieving information from a big data set. SVD represents the matrix in form of USVt. U holds information about terms, V about documents, and S is matrix for singular values. We reduce the dimension of original matrix by SVD, then perform search of queries using the most optimal truncation with minimum errors.

2 MATERIALS AND METHODS

2.1 Dataset

The dataset is a large .csv document containing 4 columns: author, posted on date, rating and text. Since information posted prior to 2009 is not taken into account, they were deleted. Number of rows decreased from about 6189 to 5144 in VSCode editor using pandas code as provided below. It deletes empty rows as well as dates prior to 2009 as shown in code below. Document is a row of ”text” column in the given .csv file. Since dataset is very large, running time of code is very long.

```python
df = pd.read_csv('comcast_consumeraffairs_complaints.csv')  #read the file, create df(table)
df = df.dropna(subset=['text']) #delete null rows
df['posted_on'] = pd.to_datetime(df['posted_on'],errors='coerce',format='mixed')
df = df[df['posted_on'].dt.year >= 2009] #delete less than 2009
df.to_csv('comcast_consumeraffairs_complaints.csv', index=False) #update csv
```

Figure 1: csv file editing

2.2 Term-by-Document Matrix

I used numpy, pandas, nltk libraries in order to create term-by-document. It is shown in figure below:

```python
import numpy as np
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
import nltk
```

Figure 2: libraries

During preprocessing, I performed tokenization, removed stopwords, stemming, and lemmatization. so now tokens contain only one term for resembling words and nonmeaningful words(stopwords) are also not counted. This significantly reduces the size of term index matrix, thus reducing complexity of all project.

```
#preprocessing text function
def preprocess_text(text):
    tokens = word_tokenize(text.lower())    #lowercasing
    tokens = [word for word in tokens if word.isalpha()]    #check alphabetic
    stop_words = set(stopwords.words('english'))    #create stopwords
    tokens = [word for word in tokens if word not in stop_words]    #remove stopwords
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(word) for word in tokens]
    return tokens

df['tokens'] = df['text'].apply(preprocess_text)
```

Figure 3: preprocessing

After preprocessing text, and creating tokens, I create and fill term-index and term document matrices as in following code. Term index matrix maps each term to one id(index). I create an empty matrix, then fill it by each cycle of loop, with elements in tokens, so that they are unique. Term document matrix shows if document has a term in it.

```
#index of each term into matrix
term_index = {}
for tokens in df['tokens']:
    for token in tokens:
        if token not in term_index:
            term_index[token] = len(term_index)

# # no of unique terms
# print("number of unique terms:", len(term_index))

# create td matrix and fill
td_matrix = np.zeros((len(term_index), len(df)))
for doc_id, tokens in enumerate(df['tokens']):
    for term in tokens:
        term_id = term_index.get(term, -1)
        if term_id != -1:
            td_matrix[term_id, doc_id] += 1
```

Figure 4: term-index and term-doc

2.3 SVD implementation

SVD is used to decompose matrix into form of USVt, so it can be reduced in size.
U is an m×m orthogonal matrix whose columns are the left singular vectors of A.
S is an m×n diagonal matrix whose non-zero elements are the singular values of A.
Vt is an n×n orthogonal matrix whose rows are the right singular vectors of A. Since we are restricted from using SVD libraries, I implement SVD using Power method. Generally, Power method is a method to find a dominant(biggest) eigenvector and largest eigenvalue. In my code, it starts with a random vector, and by iterating b=Ab/norm many times, it eventually provides approximate eigenvector.

Next function does decomposition using previous power method function. It finds most significant eigenvector and eigenvalue many times iterating num-singular-values times and fills U, S and Vt matrices. Elements of U are calculated by Av/s. The function returns U, S and Vt matrices, so decomposes original matrix.

```python
def power_method(matrix, num_iterations=100):
    b = np.random.rand(matrix.shape[1]) #random 1D vector
    for _ in range(num_iterations): #converges as iterates
        b = np.dot(matrix, b)   #A @ b
        b /= np.linalg.norm(b)  #normalize
    return b    #eigvector for

def svd_power_method(matrix, num_singular_values):
    m, n = matrix.shape
    U = np.zeros((m, num_singular_values))
    S = np.zeros(num_singular_values)
    Vt = np.zeros((num_singular_values, n))
    matrix_copy = matrix.copy()  #to preserve original matrix
    for i in range(num_singular_values):
        v = power_method(matrix_copy.T @ matrix_copy) #input to power method func is
        sigma = np.linalg.norm(matrix_copy @ v)
        u = (matrix_copy @ v) / sigma   #Av/s
        matrix_copy -= np.outer(u, v) * sigma
        U[:, i] = u #shoten
        S[i] = sigma
        Vt[i, :] = v
    return U, S, Vt
```

Figure 5: SVD using power method

2.4 Question 1

Question 1 aims to create a lower-rank approximation of original matrix with minimum loss. Lower-rank approximation is done by selecting most significant k columns of V matrix(most significant k eigenvectors), and creating new matrix called A-hat.
k is found using two methods: mean square error method and Frobenius Norm method. k is said to be optimal when MSE and FN are minimal, thus A-hat is closest to original matrix A. During iteration for both methods, k is between 10 and min(t,d)/10 and increments by 20. Formulas for mean square error and Frobenius Norm are given below.

$$MSE = \frac{1}{td}\Sigma_{i,j}(A_{ij} - \hat{A}_{ij})^2$$

$$||A - \hat{A}||_F = \sqrt{\Sigma_{i,j}(A_{ij} - \hat{A}_{ij})^2}$$

Figure 6: MSE and FN formula

According to that formula, the function of calculating MSE and FN is created as shown in code. It calculates MSE and FN for different value of k, and appends result into an array. Returns arrays containing all values of MSE and FN.

```python
def calculate_mse_and_fn(matrix, U, S, VT, k_values):
    mse_results = {}
    fn_results = {}
    for k in k_values:
        U_k = U[:, :k]  #shorten
        S_k = S[:k]  #shorten
        VT_k = VT[:k, :]  #shorten
        A_hat = U_k @ np.diag(S_k) @ VT_k   #low-approx Ahat
        mse = np.mean((matrix - A_hat) ** 2)    #formula
        fn = np.linalg.norm(matrix - A_hat) #formula
        mse_results[k] = mse    #fill array
        fn_results[k] = fn  #fill array
    return mse_results, fn_results

def find_optimal_k(mse_results, fn_results):
    optimal_k_mse = min(mse_results, key=mse_results.get)   #minimal error
    optimal_k_fn = min(fn_results, key=fn_results.get)
    return optimal_k_mse, optimal_k_fn
```

Figure 7: MSE and FN functions, optimal k function

And then using the given arrays, by finding minimal MSE and minimal FN, and its index(corresponding k), optimal k is calculated and returned.

2.5 Question 2

Question 2 requires implementing cosine similarity function, and finding most relevant document to given queries using it. When q is query vector, and d is document vector, similarity function is defined as below:

$$\textbf{Equation.10: } sim(q, d) = \frac{\vec{q} \cdot \vec{d}}{||q||.||d||}$$

Figure 8: Cosine similarity formula

```python
def cosine_similarity(v1, v2):
    norm_v1 = np.linalg.norm(v1)    #normailze q
    norm_v2 = np.linalg.norm(v2)    #normailze v
    if norm_v1 == 0 or norm_v2 == 0:
        return 0  # Return 0 similarity if either vector has zero length to avoid division by zero
    return np.dot(v1, v2) / (norm_v1 * norm_v2) #formula

def query_vector(query, term_index, U, S, k):
    q = np.zeros(len(term_index))
    for word in query:
        if word in term_index:
            q[term_index[word]] = 1 #create a query vector so it has 1 where it has terms for term-index matrix
    return q @ U[:, :k] @ np.linalg.inv(np.diag(S[:k]))

def find_most_relevant_document(query, term_index, U, S, VT, k):
    q_vec = query_vector(query, term_index, U, S, k)
    VT_reduced = VT[:k, :]  # Here VT should have rows up to k
    document_similarities = [cosine_similarity(q_vec, VT_reduced[:, i]) for i in range(VT_reduced.shape[1])] #array with cosine values
    return np.argmax(document_similarities),df.iloc[np.argmax(document_similarities)]['text'] #the more cos -> the more similar
```

Figure 9: cos-similarity, query-vector and finding document functions

In the code, i have 3 functions: cosine-similarity, query-vector and find-most-relevant-document. First, i normalize q and v vectors, then apply the formula given in Figure 8. Then in second function, create a query vector, same as term-document matrices'. So i fill 1-s where it has elements of term-index matrix.

Third function calls second function, and creates query vector. Then second argument given as d in equation is row of reduced Vt. Given q and d, we call first function, and get array or cosine values. We take maximum of it, because bigger cosine value corresponds to bigger similarity. So, third function returns index of maximum similarity and its content. After that, we call the function according to the given queries, and wait for the code to run.

3 OUTPUTS

3.1 Question 1

According to results of Question 1, optimal k is same for MSE and FN methods. It should be 490 after .csv file has been modified. MSE = 1.4249011512613608e-05 whereas FN = 25.119434398897535, which shows that SVD approximated the matrix with a little loss.

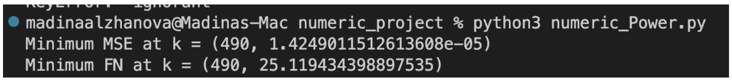

Figure 10: Results for Question 1

3.2 Question 2

Outputs of Question 2 say that query1's relevant document is at index 8, query2's is 4912, query3 and query4's is 1030 in modified dataset. The terminal screenshots are provided below.

Figure 11: Results for Question 2

In second question, the document has same words as in query, or other forms of that word or synonyms. There are some examples:

Document for query1 contains both overwhelming and ignorant.
Document for query2 contains "adapter" 4 times, also has "router" and etc.
Document for query3 contains lie(=liar), advertising(=promotion), crappy(=horrible) and etc.
Document for query4 contains intrenet, kindergartener(kindergarten), supervisor/managers(=clerk), crappy(=horrible) and etc.

4 DISCUSSION

In general, in this project I used latent semantic indexing method by using SVD. Before every process, it is crucial to reduce number of rows in dataset. Even reduced, it takes much time to run the code, so without getting rid of unnecessary texts, compile time will be too long. At first, I erased last several rows starting from 2008 year manually, but it was not enough, because file still contained empty rows and complains of older years randomly in file. So, using pandas library for this was efficient. ntlk library was used to perform preprocessing of text, and creation of term-index, term-document matrices.
In order to perform SVD I used Power method. Power method works by finding approximate eigenvector, and from that calculates eigenvalues. Having eigenvectors, we fill matrix V, with eigenvalues we calculate singualrvalues and fill diagonal matrix S. Using formula, we fill matrix U. Thus, I got SVD.
Question 1 is very useful for finding most optimal reduction size. It also can be used for compression of any text data, as well as images, so that it takes much less memory but has comparably same content. I used MSE and FN methods, and they gave same k, which means that original matrix of size about 5000 can be represented as size of 490. It reduces size more than 10 times, while its MSE is only 1.4249011512613608e-05(very negligible).
Question 2 works as a search engine. Provided queries, it finds most relevant document from large dataset. For example, google search works similarly. Searchbar gets queries, and it outputs webpages(documents) in order of relevance. In our case, we outputted only most relevant one. There are some terms in queries that are not found in a document, but it is counted as most relevant, because other terms fit the document very well. In our assignment, queries are given as terms, but they should be also preprocessed. If this code is used in larger scope, user's input must be checked for stopwords, then lemmatized, tokenized and etc.

5 CONCLUSION

In conclusion, one of efficient applications of SVD - LSI was implemented in this project. It can be used efficiently for reducing size of big dataset and to work as search engine. This project was difficult to implement because running the code takes much time, also I was not familiar with python and its libraries. But, the project was implemented successfully and results look correct.