BLG 202E - Numerical Methods in CE Project 2 Report

Nazrin Abdinli
150220925
Istanbul Technical University
Artificial Intelligence and Data Engineering
Email: abdinlin22@itu.edu.tr

***Abstract*—This report outlines our application of Latent Semantic Indexing (LSI) through Singular Value Decomposition (SVD) on a dataset comprising complaints directed at Comcast Corporation. LSI, a method in natural language processing and information retrieval, explores the connections between terms and documents in a corpus. By depicting terms and documents in a reduced-dimensional semantic framework, LSI uncovers latent semantic associations, thus refining information retrieval and analysis. The project aims to uncover meaningful insights from consumer feedback data, ultimately enhancing customer satisfaction and improving business operations within Comcast Corporation.**

I. Introduction

Latent Semantic Indexing (LSI) is a potent technique utilized in natural language processing and information retrieval to dissect the inherent semantic framework of textual data. Through the depiction of documents and terms in a reduced-dimensional semantic realm, LSI facilitates the revelation of concealed semantic connections, thereby enabling more efficient information retrieval and analysis. This report outlines the deployment of LSI employing Singular Value Decomposition (SVD) on a collection of consumer complaints directed towards Comcast Corporation. The project aims to uncover meaningful insights from consumer feedback data, ultimately enhancing customer satisfaction and improving business operations within Comcast Corporation.

II. Description of the Project

This project aims to apply Latent Semantic Indexing (LSI) through Singular Value Decomposition (SVD) on a dataset containing grievances lodged against Comcast Corporation by consumers. The dataset includes complaint information like author, date of posting, satisfaction rating, and the complaint itself. The implementation involves various essential stages:

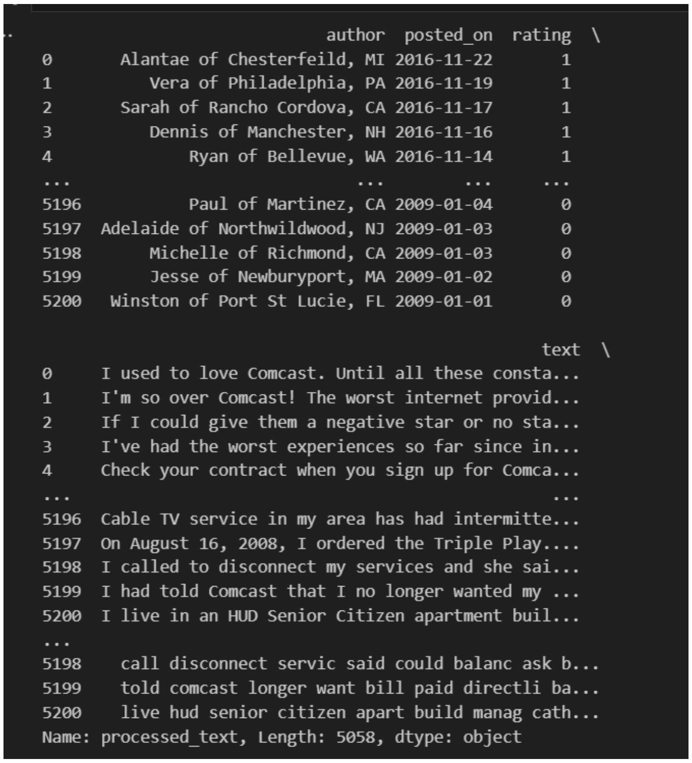
Libraries that are used for the project is below:

```
#libraries
import numpy as np
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, ↩
    WordNetLemmatizer
from sklearn.feature_extraction.text import ↩
    TfidfVectorizer
```

A. Question 1

1) Data Loading and Preprocessing: The consumer complaints dataset is loaded and preprocessed to filter complaints from the year 2009 onwards and remove any rows with missing complaint details.

```
#loading the dataset
df = pd.read_csv("↩
    comcast_consumeraffairs_complaints.csv")

df['posted_on'] = pd.to_datetime(df['↩
    posted_on'])
df_filter_2009 = df[df['posted_on'].dt.year ↩
    >= 2009].copy()
df_filter_2009.dropna(subset=['text'], ↩
    inplace=True)

#text preprocessing
stop_words = set(stopwords.words('english'))
port_stem = PorterStemmer()
wn_lemmatizer = WordNetLemmatizer()

def preprocess_text(text):
    tokens = word_tokenize(text.lower())
    tokens = [token for token in tokens if ↩
        token.isalnum()]
    tokens = [token for token in tokens if ↩
        token not in stop_words]
    tokens = [port_stem.stem(token) for ↩
        token in tokens]
    tokens = [wn_lemmatizer.lemmatize(token)↩
         for token in tokens]
    return " ".join(tokens)

df_filter_2009['processed_text'] = ↩
    df_filter_2009['text'].apply(↩
    preprocess_text)

print(df_filter_2009)
print(df_filter_2009['processed_text'])
```

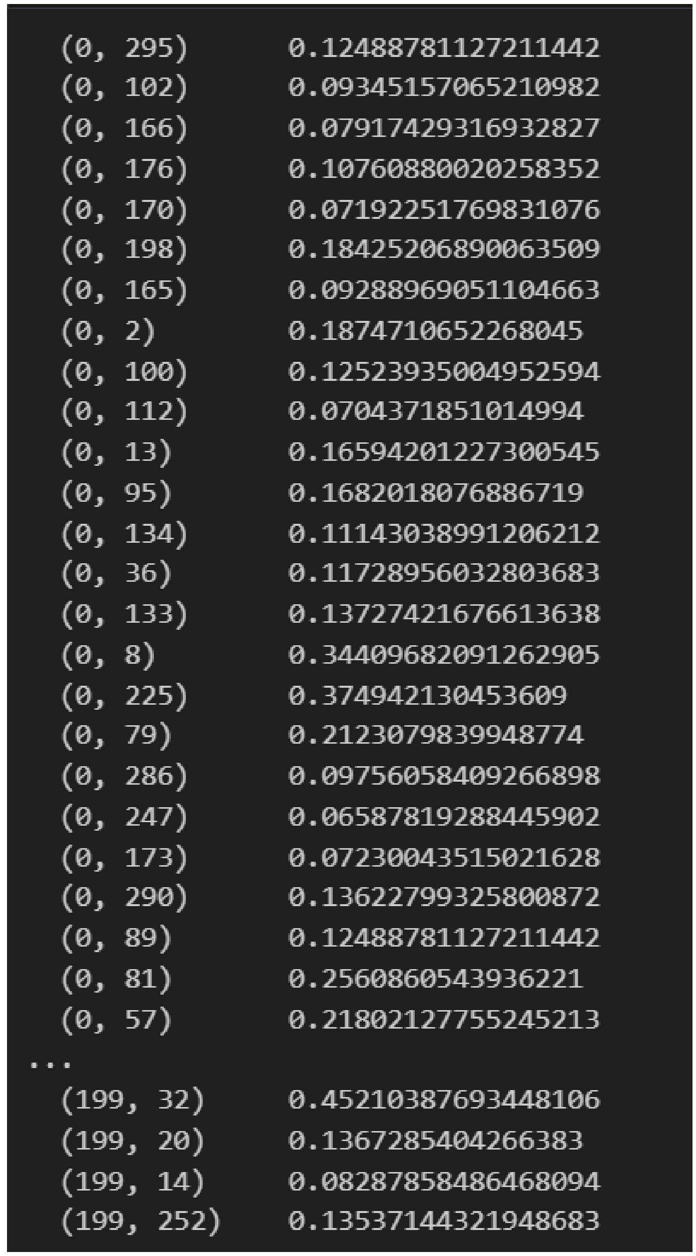


```
                         author  posted_on  rating  \
0      Alantae of Chesterfeild, MI 2016-11-22       1
1         Vera of Philadelphia, PA 2016-11-19       1
2     Sarah of Rancho Cordova, CA 2016-11-17       1
3         Dennis of Manchester, NH 2016-11-16       1
4             Ryan of Bellevue, WA 2016-11-14       1
...                            ...        ...     ...
5196          Paul of Martinez, CA 2009-01-04       0
5197  Adelaide of Northwildwood, NJ 2009-01-03       0
5198       Michelle of Richmond, CA 2009-01-03       0
5199       Jesse of Newburyport, MA 2009-01-02       0
5200    Winston of Port St Lucie, FL 2009-01-01       0

                                                   text  \
0     I used to love Comcast. Until all these consta...
1     I'm so over Comcast! The worst internet provid...
2     If I could give them a negative star or no sta...
3     I've had the worst experiences so far since in...
4     Check your contract when you sign up for Comca...
...                                                 ...
5196  Cable TV service in my area has had intermitte...
5197  On August 16, 2008, I ordered the Triple Play....
5198  I called to disconnect my services and she sai...
5199  I had told Comcast that I no longer wanted my ...
5200  I live in an HUD Senior Citizen apartment buil...
...
5198    call disconnect servic said could balanc ask b...
5199    told comcast longer want bill paid directli ba...
5200    live hud senior citizen apart build manag cath...
Name: processed_text, Length: 5058, dtype: object
```

Fig. 1. Filtered Dataset

2) Term-by-Document Matrix Construction: A term-by-document matrix is constructed from the preprocessed complaint text, where each row represents a document (complaint) and each column represents a term. Due to big size of matrix, sample data from original dataset is used for checking the code.

```
#initializing TfidfVectorizer with desired ↩
    parameters
vectorizer = TfidfVectorizer()

#fitting and transforming the sample data
term_doc_matrix = vectorizer.fit_transform(↩
    df_filter_2009['processed_text'])
print(term_doc_matrix)

#getting sample data from original data for ↩
    checking the code
sample_data = df_filter_2009["text"][:200]

vectorizer = TfidfVectorizer(max_features↩
    =300)

term_doc_matrix = vectorizer.fit_transform(↩
    sample_data)
```



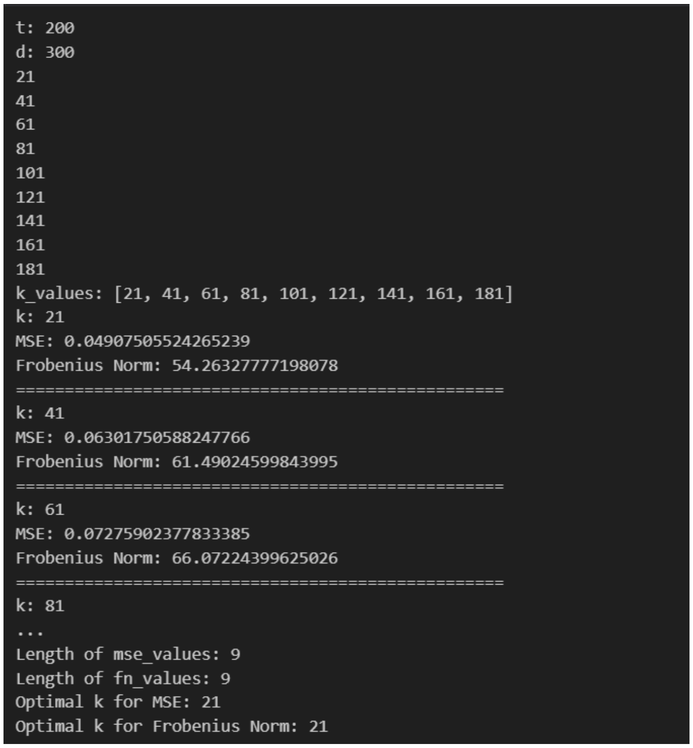
```
  (0, 295)	0.12488781127211442
  (0, 102)	0.09345157065210982
  (0, 166)	0.07917429316932827
  (0, 176)	0.10760880020258352
  (0, 170)	0.07192251769831076
  (0, 198)	0.18425206890063509
  (0, 165)	0.09288969051104663
  (0, 2)	0.1874710652268045
  (0, 100)	0.12523935004952594
  (0, 112)	0.0704371851014994
  (0, 13)	0.16594201227300545
  (0, 95)	0.1682018076886719
  (0, 134)	0.11143038991206212
  (0, 36)	0.11728956032803683
  (0, 133)	0.13727421676613638
  (0, 8)	0.34409682091262905
  (0, 225)	0.374942130453609
  (0, 79)	0.2123079839948774
  (0, 286)	0.09756058409266898
  (0, 247)	0.06587819288445902
  (0, 173)	0.07230043515021628
  (0, 290)	0.13622799325800872
  (0, 89)	0.12488781127211442
  (0, 81)	0.2560860543936221
  (0, 57)	0.21802127755245213
...
  (199, 32)	0.45210387693448106
  (199, 20)	0.1367285404266383
  (199, 14)	0.08287858486468094
  (199, 252)	0.13537144321948683
```

Fig. 2. Term-by-Document Matrix of Sample Data

3) SVD Computation and Implementation: SVD is applied to the term-by-document matrix to reduce its dimensionality and uncover latent semantic relationships between terms and documents. The quality of the SVD approximation is evaluated using Mean Squared Error (MSE) and Frobenius Norm (FN) metrics, allowing us to determine the optimal number of dimensions for the reduced semantic space.

```
#custom SVD function
def svd(A):
    Ui = A.dot(A.transpose())
    Vi = A.transpose().dot(A)
    _,eig_values_U, U = custom_eig(Ui)
    U = U.real
    eig_values_U.real
    _,eig_values_V, V = custom_eig(Vi)
    V = V.real
    eig_values_V.real
    # Sorting eigenvalues and corresponding eigenvectors in descending order
    idx_U = np.argsort(eig_values_U)[::-1]
    idx_V = np.argsort(eig_values_V)[::-1]
    U = U[:, idx_U]
    V = V[:, idx_V]
    # Taking square root of positive eigenvalues
    Si = np.sqrt(np.maximum(eig_values_U, 0))
    S = np.diag(Si)
    return U, S, V.transpose()

#custom eigenvalue decomposition function
def custom_eig(matrix, epsilon=1e-10, max_iterations=1000):
    matrix_dense = matrix.toarray()  # Convert to dense array
    m, n = matrix_dense.shape
    eigenvalues = np.zeros(n)
    eigenvectors = np.eye(n)
    U = np.eye(m)  # Initialize U as a 2D identity matrix
    for i in range(n):
        v = np.random.rand(n)
        for _ in range(max_iterations):
            v_next = np.dot(matrix_dense, v)
            v_next_norm = np.linalg.norm(v_next)
            v_next /= v_next_norm
            eigenvalue = np.dot(v_next, np.dot(matrix_dense, v_next))
            if np.abs(eigenvalue - eigenvalues[i]) < epsilon:
                break
            v = v_next
            eigenvalues[i] = eigenvalue
        eigenvectors[:, i] = v_next
        U[:, i] = v  # Update U with the computed eigenvector
        matrix_dense -= eigenvalues[i] * np.outer(v_next, v_next)
    return eigenvectors, eigenvalues, U

#computing SVD
U, Sigma, V_T = svd(term_doc_matrix)

#printing shapes of resulting matrices
print("Shape of U:", U.shape)
print("Shape of Sigma:", Sigma.shape)
print("Shape of V_T:", V_T.shape)

#evaluating SVD Approximation
def calculate_mse(original_matrix, reconstructed_matrix):
    #MSE
    mse = np.mean(np.square(original_matrix - reconstructed_matrix))
    return mse

def calculate_frobenius_norm(original_matrix, reconstructed_matrix):
    #frobenius norm
    fn = np.linalg.norm(original_matrix - reconstructed_matrix, ord='fro')
    return fn

#printing dimensions of term_doc_matrix
t, d = term_doc_matrix.shape
print("t:", t)
print("d:", d)

min_k = max(10, min(t, d) // 10 + 1)
k_values = []
for i in range(min_k, min(t, d) + 1, 20):
    print(i)
    k_values.append(i)

print("k_values:", list(k_values))

mse_values = []
fn_values = []

for k in k_values:
    U_k, Sigma_k, V_T_k = svd(term_doc_matrix)
    U_k = U_k[:, :k]
    Sigma_k = Sigma_k[:k, :k]
    V_T_k = V_T_k[:, :k]
    reconstructed_matrix = np.dot(np.dot(U_k, Sigma_k), V_T_k.transpose())

    #calculating Mean Squared Error (MSE) and Frobenius Norm (FN)
    mse = calculate_mse(term_doc_matrix, reconstructed_matrix)
    fn = calculate_frobenius_norm(term_doc_matrix, reconstructed_matrix)
    mse_values.append(mse)
    fn_values.append(fn)

    print("k:", k)
    print("MSE:", mse)
    print("Frobenius Norm:", fn)
    print("="*50)

#checking if MSE and FN lists are empty
if not mse_values:
    print("Error: mse_values is empty.")
if not fn_values:
    print("Error: fn_values is empty.")

print("Length of mse_values:", len(mse_values))
print("Length of fn_values:", len(fn_values))
```

```
#finding optimal k values for MSE and FN
optimal_k_mse = k_values[np.argmin(↩
    mse_values)]
optimal_k_fn = k_values[np.argmin(fn_values)↩
    ]

print("Optimal k for MSE:", optimal_k_mse)
print("Optimal k for Frobenius Norm:", ↩
    optimal_k_fn)
```



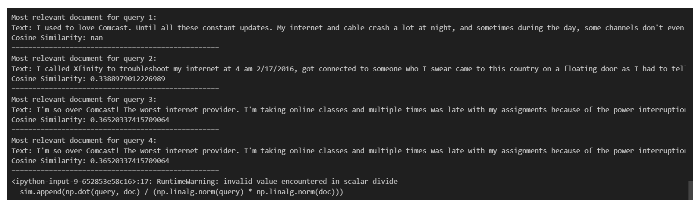
```
t: 200
d: 300
21
41
61
81
101
121
141
161
181
k_values: [21, 41, 61, 81, 101, 121, 141, 161, 181]
k: 21
MSE: 0.04907505524265239
Frobenius Norm: 54.26327777198078
==================================================
k: 41
MSE: 0.06301750588247766
Frobenius Norm: 61.49024599843995
==================================================
k: 61
MSE: 0.07275902377833385
Frobenius Norm: 66.07224399625026
==================================================
k: 81
...
Length of mse_values: 9
Length of fn_values: 9
Optimal k for MSE: 21
Optimal k for Frobenius Norm: 21
```

Fig. 3. SVD Computation and Implementation for Sample Data

B. Question 2

1) Query-Document Cosine Similarity: Cosine similarity is calculated between user queries and documents to retrieve the most relevant documents for each query, facilitating effective information retrieval and analysis.

```
#query-document cosine similarity
queries = [
    ['ignorant', 'overwhelming'],
    ['xfinity', 'frustrate', 'adapter', '↩
        verizon', 'router'],
    ['terminate', 'rent', 'promotion', 'joke↩
        ', 'liar', 'internet', 'horrible'],
    ['kindergarten', 'ridiculous', 'internet↩
        ', 'clerk', 'terrible']
]

#calculating TF-IDF matrix for queries
query_matrix = vectorizer.transform([' '.↩
    join(query) for query in queries]).↩
    toarray()

#calculating cosine similarity
cosine_similarities = []
for query in query_matrix:
    sim = []
    for doc in term_doc_matrix.toarray():
        sim.append(np.dot(query, doc) / (np.↩
            linalg.norm(query) * np.linalg.↩
            norm(doc)))
    cosine_similarities.append(sim)

cosine_similarities = np.array(↩
    cosine_similarities)

#finding the most relevant document for each↩
     query
for i, query in enumerate(queries):
    most_similar_doc_index = np.argmax(↩
        cosine_similarities[i])
    most_similar_doc_text = df_filter_2009.↩
        iloc[most_similar_doc_index]['text']
    print(f"Most relevant document for query↩
         {i+1}:")
    print("Text:", most_similar_doc_text)
    print("Cosine Similarity:", ↩
        cosine_similarities[i, ↩
        most_similar_doc_index])
    print("="*50)
```



```
Most relevant document for query 1:
Text: I used to love Comcast. Until all these constant updates. My internet and cable crash a lot at night, and sometimes during the day, some channels don't even
Cosine Similarity: nan
==================================================
Most relevant document for query 2:
Text: I called Xfinity to troubleshoot my internet at 4 am 2/17/2016, got connected to someone who I swear came to this country on a floating door as I had to tel
Cosine Similarity: 0.3388979012226989
==================================================
Most relevant document for query 3:
Text: I'm so over Comcast! The worst internet provider. I'm taking online classes and multiple times was late with my assignments because of the power interruptio
Cosine Similarity: 0.36520337415709064
==================================================
Most relevant document for query 4:
Text: I'm so over Comcast! The worst internet provider. I'm taking online classes and multiple times was late with my assignments because of the power interruptio
Cosine Similarity: 0.36520337415709064
==================================================
<ipython-input-9-652853e58c16>:17: RuntimeWarning: invalid value encountered in scalar divide
  sim.append(np.dot(query, doc) / (np.linalg.norm(query) * np.linalg.norm(doc)))
```

Fig. 4. Query-Document Cosine Similarity for Sample Data

III. Conclusion

In conclusion, this project demonstrates the effectiveness of Latent Semantic Indexing (LSI) using Singular Value Decomposition (SVD) in analyzing consumer complaints data against Comcast Corporation. Through LSI methods, the project revealed concealed semantic connections within the data, offering valuable perspectives into customer feedback and refining information retrieval procedures and its outcomes underscore LSI's capacity to enhance customer contentment, guide business strategies, and foster enhancements within Comcast Corporation.