Search within a collection of documents

Mathematical Modelling

Nik Jenič, Tian Ključanin, Maša Uhan

June 9, 2024

Contents

1 Introduction 3

2 Search within a collection of documents 4
2.1 Approach and Methodology 4
2.1.1 Data Collection 4
2.1.2 Implementing the LSI model 4
2.1.3 Improving the model 5

3 Solution 6
3.1 Frequency Solution 6
3.2 Weighted Solution 6
3.3 Code Utilization and Output: 7
3.4 Additional Improvements 7
3.4.1 Adding New Documents 7
3.4.2 Adding New Words 8

4 Implementation 9
4.1 Tools 9
4.1.1 NumPy 9
4.1.2 Scikit-learn 9
4.1.3 Python-Docx 10
4.2 Testing Implementation and Methodology 10
4.2.1 Automatic Testing 10
4.2.2 Manual Testing 10
4.2.3 Evaluation Criteria 11

5 Results 12
5.1 Automatic testing 12
5.2 Manual testing 13
5.3 Discussion 14
5.3.1 Comparing Original and Weighted solution 14
5.3.2 Folding-in new documents and terms 15

6 Conclusion 16

7 References and Code 17
7.1 main.py 17
7.2 generateMatrix.py 26
7.3 debug.py 30

1 Introduction

In today's digital landscape, the abundance of online information poses a significant challenge known as information overload. Traditional search methods, relying heavily on exact keyword matches, often struggle to cope with this deluge of data. They fail to account for the diverse ways people express ideas, such as using synonyms and related terms, leading to incomplete or irrelevant search results.

This deficiency highlights the need for a more sophisticated approach that can decipher the deeper semantic relationships between words and documents. Latent Semantic Indexing (LSI) offers a solution by going beyond literal keyword matching. It builds a model which understands the conceptual connections within content, thus improving the accuracy and comprehensiveness of information retrieval.

2 Search within a collection of documents

This project aims to implement an LSI-based search engine that can efficiently process and retrieve relevant documents from a collection based on user queries.

2.1 Approach and Methodology

The project involves the following key steps:

1. Data Collection: Gather a collection of documents to form the basis of the search engine.
2. Implementing the LSI model: Develop an LSI model to analyze the relationships between words and documents in the collection.
3. Testing and improving the model.

2.1.1 Data Collection

We are using publicly available data for analysis. This involves curating a diverse set of documents to ensure the search engine's effectiveness across different topics and domains.

2.1.2 Implementing the LSI model

Implementing the LSI model entails the following steps:

1. Building an A matrix of connections between words and documents from a document selection, where each document has its own column in the matrix and each word has its row. The element a_{ij} represents the frequency of the i-th word in the j-th document.
2. Splitting the matrix A using the SVD method, where $A = U_k S_k V_k^T$, which only has k significant singular values.
3. Creating a query vector q from the query, where each element represents the frequency of a word in the query. We only consider words that appear in the document collection.
4. Generating a new vector from the q query vector in the document space with the formula $\hat{q} = q^T U_k S_k^{-1}$. The query should return documents for which the cosine similarity value is higher than the selected limit.

2.1.3 Improving the model

The model can be improved by replacing the frequencies in matrix A with more complex measurements. In general, the element of the matrix can be written as a product:

$$a_{ij} = L_{ij} \cdot G_i$$

where L_{ij} is the local measure of the importance of a word in a document, and G_i is the global measure of the importance of a word.

In this project, we are using a scheme where the local measure of importance is given by the logarithm of the frequency f_{ij} of the i-th word in the j-th document:

$$L_{ij} = \log(f_{ij} + 1)$$

and the global measure of importance is calculated using entropy:

$$G_i = 1 - \sum_j \frac{p_{ij} \log(p_{ij})}{\log n}$$

where n is the total number of documents in the collection, $p_{ij} = \frac{f_{ij}}{gf_i}$, and gf_i is the frequency of a word in the whole collection.

The model can be further improved by adding new documents or words without having to recalculate the SVD of the matrix A.

3 Solution

3.1 Frequency Solution

Our initial approach involves the construction of a basic frequency matrix, which serves as the foundation for text representation within our system. We construct the matrix as described below:

- **Word Collection:** Initially, we gather all unique words from the corpus, disregarding duplicates. This collection forms the basis of the rows in our matrix, with each word allocated a specific row.
- **Document Parsing:** Each document in the dataset is processed to extract the words it contains. These documents correspond to the columns of our matrix.
- **Frequency Calculation:** For each document, we count the occurrences of each word and populate the matrix accordingly. The intersection of a row and a column in the matrix holds the frequency of the word (row) in the specified document (column).
- **Matrix Assembly:** The complete matrix is assembled by combining the word frequencies across all documents. This matrix is then utilized to represent the text data in a structured form.

This matrix construction does not include any advanced data handling or algorithmic optimization but lays the groundwork for further processing and analysis. The simplicity of this method gives us a clear view of the text data's key elements.

3.2 Weighted Solution

Looking further to improve our search methods, a key challenge lies in managing the influence of word frequencies. Common words can dominate search results, while rare words might disproportionately affect outcomes, even if they're informative. To solve this problem, we have implemented an optimization technique for the term-document matrix that helps ensure no single group of words skews the results too much.

Our weighted solution improves previous method by substituting word frequencies in our matrix with more sophisticated metrics. These include a local measure that employs logarithmic transformations of word frequencies, and a global measure derived from entropy. This approach ensures that the impact of each word on search results accurately reflects its genuine informational value while maintaining a balanced and effective retrieval system.

3.3 Code Utilization and Output:

The code operates by taking a collection of documents as input and proceeds through several steps:

1. **Matrix Generation:** All text documents are processed to generate the basic frequency matrix as described above.
2. **Document Similarity Analysis:** Using Singular Value Decomposition (SVD), the system identifies and quantifies the similarity between the textual content of the documents based on the transformed matrix data.
3. **Query Handling:** The system allows users to input a query, which is then converted into a vector. This vector is used to find documents that are most similar to the query, based on cosine similarity metrics.

The primary outputs from this process include a list of documents ranked by their relevance to the input query. These results help identify the most pertinent documents without the need to manually sift through the entire dataset.

3.4 Additional Improvements

So far, our approach, while efficient, still requires recomputing the Singular Value Decomposition (SVD) each time a new document or word is added. To resolve this issue, we have implemented an update mechanism that integrates new data into the existing SVD structure without the need for full recomputation. This enhancement significantly improves the scalability and efficiency of our system, enabling more effective handling of dynamic datasets.

3.4.1 Adding New Documents

When adding a new document, we first create a vector q for the document, where each entry represents the frequency of a word from that document. This only includes words, which have previously appeared in other documents, skipping any new words, which will be added later. This vector is then transformed into the existing document space using the formula:

$$\hat{q} = q^T U_k S_k^{-1}$$

The vector is then appended to the matrix V_k:

$$\hat{V_k} = \begin{bmatrix} V_k & \hat{q} \end{bmatrix}$$

3.4.2 Adding New Words

In the process of updating the SVD with new documents, a frequency vector is constructed for each new word encountered. This vector is initially populated with zeros for all existing documents, reflecting the absence of the new word in those texts. The frequency count of the new word in the new document is then recorded in the last entry of this vector. This vector q is then transformed into the document space using the formula:

$$\hat{q} = qV_k^T S_k$$

Then, we append the vector to the matrix U_k:

$$\hat{U_k} = \begin{bmatrix} U_k \\ \hat{q} \end{bmatrix}$$

4 Implementation

With our understanding of the problem and the proposed solutions, we now move to the implementation phase. This section details the technical aspects of our system, including the code structure, key functions, and the overall workflow.

4.1 Tools

The first step in implementing our system is to select the appropriate tools and programming languages.

We decided to use the programming language *Python*, due to its ease of use, versatility, and extensive collection of libraries. The following libraries are essential for our implementation:

- NumPy
- Scikit-learn
- Python-Docx

4.1.1 NumPy

The leading role in our implementation is played by NumPy, a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

Since our system relies heavily on large matrix operations, NumPy is an essential library for handling the underlying data structures and computations.

4.1.2 Scikit-learn

Scikit-learn is a powerful machine learning library that provides simple and efficient tools for data mining and data analysis.

We utilize Scikit-learn for its implementation of the SVD algorithm, which is a key component of our LSI model. The SVD implementation we use is called "randomized_svd", which is particularly well-suited for large sparse datasets, like the one we are working with. This implementation allows us to efficiently find a close approximation of the truncated SVD, which is crucial for our system's performance.

Another important feature of Scikit-learn that we leverage is their collection of machine learning datasets. The dataset we decided to use for testing and evaluation purposes is the "20 Newsgroups" dataset, which contains approximately 13,000 newsgroup documents across 20 different categories.

4.1.3 Python-Docx

A simple library that we used for reading DOCX files. This allowed us to easily test our implementation on a smaller local dataset.

4.2 Testing Implementation and Methodology

To ensure the accuracy and efficiency of our document retrieval system, thorough testing was conducted using a combination of automatic and manual methods. These methods were designed to address both the performance and the dynamic capabilities of the system under various conditions.

4.2.1 Automatic Testing

Our test data is structured in the form of emails, each including a "subject" line which serves as a concise description of the content. In our testing setup, we removed the subject line from the email to use as a query, reducing bias in testing. The system then retrieves the most relevant emails based on these queries. Results are compared to the original subject lines to assess relevance. The scoring system awards full points if the correct email is the most similar to the query. If the correct email ranks within the top ten of the most similar emails, a full point is awarded (our testing shows that similarity score may be quite low even for most similar documents, so a full point is used regardless of the similarity score).

4.2.2 Manual Testing

In addition to automated tests, we manually selected queries and assessed the relevance of the returned documents. This approach allowed for a detailed evaluation of the system's effectiveness and helped identify potential areas for improvement. Different metrics were utilized to assess whether a returned document was genuinely relevant.

4.2.3 Evaluation Criteria

The evaluation process therefore includes several key steps to ensure comprehensive testing:

1. **Relevance and Accuracy Assessment:** Documents are retrieved based on cosine similarity measures for each query. Accuracy is quantified by the closeness of the retrieved documents to predefined relevant documents or subjects.

2. **Performance Scoring:** The system assigns scores based on the relevance of the retrieved documents. Exact matches receive full points, while partial scores are awarded for close matches, based on their similarity metrics.

3. **Dynamic Data Integration Tests:** The system's ability to dynamically update the term-document matrix and SVD components when new documents are added is critically tested. This ensures that the system maintains its accuracy and efficiency as the dataset grows.

This structured testing strategy ensures that our document retrieval system is thoroughly evaluated for accuracy and reliability.

5 Results

5.1 Automatic testing

Here is an overview of the table structures used to present the results of our testing process. These tables provide a clear and concise summary of the system's performance under different conditions, allowing for easy comparison and analysis of the results. The tables are structured as follows:

- **Rows:** Each row represents a different k value, where k is the number of biggest singular values in the SVD decomposition. The values range from 10 to 1000, allowing us to test the system's scalability across different dimensions of space.
- **Columns:** The columns represent different threshold values, related to the cosine similarity thresholds used in the retrieval process. These values range from 0.1 to 0.9, affecting how strict the similarity criterion is for considering two documents as closely related.
- **Cell Values:** Each cell in the table represents the number of points the program scored for each combination of k and cosine similarity threshold. These values illustrate how the retrieval effectiveness varies with the complexity of the model and the strictness of the similarity threshold.

The tables present the results of testing different configurations of the non-weighted and weighted solution implemented in the document retrieval system (results based on testing on 1000 documents).

	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
10	52.17	52.17	52.17	52.17	52.17	51.59	50.97	44.25	25.77
50	182.6	182.6	182.6	177.4	157.5	115.1	93.88	73.63	30
100	277.3	277.3	271.3	237.5	182.5	122.6	100.7	70	22
250	469.2	468.2	436.3	355.4	255.8	191.9	127.7	83	26
500	613.5	606.9	557.6	465.0	356.9	239.6	155	81	28
750	668.4	660.4	618.1	526.5	389.5	281.7	168	100	26
1000	641.9	627.2	580.6	486.4	385.2	295.6	213	119	37

Table 1: Non-weighted Solution for Different Values of k

	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
10	71.18	71.18	71.18	71.18	70.72	68.47	68.47	63.06	44.12
50	300.6	300.6	300.0	298.9	286.6	259.3	198.7	128.5	57.95
100	442.1	442.1	441.2	417.0	354.4	273.8	180.7	116.5	58
250	637.2	636.5	622.4	547.8	426.8	304.0	213.5	128.8	61
500	726.5	722.5	674.8	576.7	435.6	325.3	201.0	102	65
750	753.3	741.5	685.0	589.2	459.6	322.3	219	123	56
1000	673.5	657.3	604.0	499.4	409.7	316.6	232	141	70

Table 2: Optimized Solution for Different Values of k

5.2 Manual testing

We manually tested the weighted solution with the improvement of folding-in new documents and words. This was done on an weighted solution. This was done on a collection of 1000 documents and we slowly added more documents and terms to the system, till it's behaviour became unpredictable and the results seemed random.

5.3 Discussion

In this section we will discuss our findings.

5.3.1 Comparing Original and Weighted solution

Observing the results from both the non-weighted and weighted solutions, several key insights can be drawn:

- The weighted solution consistently outperforms the non-weighted solution across all tested configurations. This improvement is most notable in scenarios with higher k values and stricter cosine similarity thresholds.
- The weighted solution demonstrates better scalability and robustness, maintaining higher scores as the dimensionality of the space increases.
- The non-weighted solution shows a decline in performance as the number of singular values k grows, indicating limitations in handling larger datasets and more complex models.
- Both solutions exhibit a general trend of decreasing scores with stricter cosine similarity thresholds, reflecting the trade-off between precision and recall in document retrieval.

We conducted different testing for the weighted solution, one of which was changing the formula for the global measure of importance. We changed the formula from:

$$G_i = 1 - \sum_j \frac{p_{ij}\log(p_{ij})}{\log n} \quad \longrightarrow \quad G_i = 1 + \sum_j \frac{p_{ij}\log(p_{ij})}{\log n}$$

This causes the inverse effect of the original goal, which was preventing ”overfitting”, meaning putting too much focus on specific words.

The results showed that the weighted solution with the original formula performed better - this may be due to key words often found in the subject line in our documents.

5.3.2 Folding-in new documents and terms

We have noticed that when folding-in new terms and documents, the system's performance starts declining rapidly after adding around 5% new documents (relatively to the number of already existing ones). This is due to the fact that the new documents are not as relevant to the existing ones, and the system has a hard time finding the right documents.

This is a common issue with LSI models, as they are not as effective when dealing with large amounts of new data that is not closely related to the existing data.

Folding-in more than 5% new documents and/or terms there is a rapid fall off in performance, and behaviour of the system may become unpredictable. The results we got seemed random and we advise against using the system in such a state.

6 Conclusion

Through making this paper, we found that LSI models are an effective tool for searching through large collections of documents, with a relatively simple implementation. We have also found that the weighted solution outperforms the non-weighted solution, though different weighing methods are worth looking into. The system can also be improved by adding new documents and terms without having to recalculate the SVD of the matrix, which could lead to long downtimes in a real-world scenario.

This project helped us in the field of machine learning and natural language processing, while also reinforcing our programming and linear algebra knowledge.

7 References and Code

All the code used in this project can be found on our GitHub repository: `https://github.com/TianK003/MM_Projektna_Naloga` under the folder ”Koda”.

7.1 main.py

```
import numpy as np
import generateMatrix as gm
import debug
import sys
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
from sklearn.utils.extmath import randomized_svd
import argparse
import os
import shutil
import time

# NOTES for later:
# Matrix of words: Every document is a column, every word is a row
# For optimized search, we use a hashmap to store the index of the word in
    the matrix

document_folder = "documents"
k = 1000
cosinus_threadhold = 0.3
modes = ["unoptimized", "optimized"]
mode = modes[0]
data_limit = 1000000000
new_data_limit = 1000000000

save_files_folder = os.path.join("savedMatricies", "savedMatricies_") #
    Appended in argsparse
u_file = "U.npy"
s_file = "s.npy"
v_file = "V.npy"
file_names_file = "file_names.npy"
word_map_file = "word_map.npy"
word_list_file = "word_list.npy"
matrix_file = "matrix.npy"
recompute = False
should_add_new_documents = False
testing = False

testing_score = 0 # For each correct document +1, for each document that was
     close but not correct + similarity score


def matrix_to_file(matrix, file_name):
    # For debugging purposes, write the matrix to a file
    f = open(file_name, 'w')
    f.write(np.array2string(matrix, threshold=np.inf, max_line_width=np.inf)
        )
    f.close()

def format_print_text(text, i):
    # Make all lines the same length, no longer than the terminal window,
        and add dots to every second line
    print_text = text
    max_size = 60
    terminal_size = os.get_terminal_size().columns-13
    if terminal_size > max_size:
        terminal_size = max_size
    if len(print_text) > terminal_size:
        print_text = print_text[:terminal_size-3] + "..."
    if len(print_text) < terminal_size:
        if (len(print_text) + 1)%2 == 0:
            print_text += "␣"
        space_symbol = "␣."
        if i%2 == 0:
            space_symbol = "␣␣"
        print_text += space_symbol*int((terminal_size-len(print_text))/2)

    return print_text

def compute_svd(matrix, k): # We use the sklearn library to compute the SVD,
     as it's faster and more efficient for sparse matrices
    debug.log("Performing␣SVD")
    U, s, V = randomized_svd(matrix, n_components=k)
    debug.log("SVD␣performed")

    return U, s, V

def svd(matrix, k, files_saved): # Singular Value Decomposition

    if files_saved: # If we have the data saved, we can just load it
        debug.log("Loading␣SVD")
        U = np.load(os.path.join(save_files_folder, u_file))
        s = np.load(os.path.join(save_files_folder, s_file))
        V = np.load(os.path.join(save_files_folder, v_file))
        debug.log("SVD␣loaded")
    else:
        debug.log("Computing␣SVD")
        U, s, V = compute_svd(matrix, k)

        debug.log("Saving␣SVD")
        np.save(os.path.join(save_files_folder, u_file), U)
        np.save(os.path.join(save_files_folder, s_file), s)
        np.save(os.path.join(save_files_folder, v_file), V)

    if k > len(s):
        k = len(s)
    if k > data_limit:
        k = data_limit
    debug.log("k:␣" + str(k))
    u_k = U[:, :k]
    s_k = np.diag(s[:k])
    v_k = V[:k, :]

    debug.log("shape␣of␣v_k␣here:␣" + str(v_k.shape))

    return u_k, s_k, v_k

def build_query_vector(prompt, word_map, word_list):
    # Create a query vector from the prompt
    # debug.log("Building query vector")
    query_vector = np.zeros(len(word_list))
    words = prompt.split()
    for word in words:
        if word in word_map:
            query_vector[word_map[word]] += 1
    return query_vector

def s_k_inverse(s_k):
    # Invert the diagonal matrix s_k
    inverteds_k = np.zeros(s_k.shape)
    for i in range(s_k.shape[0]):
        if s_k[i, i] != 0:
            inverteds_k[i, i] = 1 / s_k[i, i]
    return inverteds_k

def cosine_similarity(v1, v2):
    # Find the angle between two vectors
    if not np.any(v1) or not np.any(v2):
        return 0
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))

def find_closest_documents(q_altered, v_k, file_names):
    # Find the closest documents to the query vector
    # q_altered is the query vector altered by the SVD
    # v_k is the V matrix from the SVD
    # debug.log("Finding closest documents")

    # Compare the search vector to all the document vectors based on cosine
        similarity
    closest_documents = []
    for i in range(v_k.shape[1]):
        v = v_k[:, i]
        similarity = abs(cosine_similarity(q_altered, v))
        if similarity >= cosinus_threadhold:
            closest_documents.append([i, similarity])

    # Sort the documents by similarity, highest similarity first
    closest_document_names = [[file_names[i],similarity] for i,similarity in
         closest_documents]
    closest_document_names = sorted(closest_document_names, key=lambda x: x
        [1])[::-1]

    return closest_document_names

def testing_analysis(closest_documents, subject, temp_file_names, temp_data)
    :
    global testing_score
    # debug.log("Testing analysis")
    for i in range(len(closest_documents)):
        if i >= 10:
            break

        # Since we only have the index of the document in the V matrix, we
            need to find the corresponding title and data. So we search for
            the index in the list of titles
        # closest_documents[i][0] is the title of the document, so we find
            the index of the title in the list of titles
        index = np.nonzero(np.array(temp_file_names) == closest_documents[i
            ][0])[0][0]
        current_data = temp_data[index]
        gotten_subject = gm.get_subject_from_document(current_data).lower()
        if gotten_subject == subject: # If the "correct" document was within
             the top 10, add the similarity score. If it was the most
            similar, add 1. This testing method rewards overfitting, which
```

```
                might not be ideal, depending on the wanted result
            testing_score += 1
            return

    debug.log("Failed␣to␣find␣subject:␣|" + subject + "|")

def analyze_results(closest_documents):
    # Print the results of the search

    if len(closest_documents) == 0:
        print("\033[94mNo␣documents␣found␣\033[0m")
        return

    while True:
        print("\n")
        print("\033[94mClosest␣documents:␣\033[0m")
        i = 0
        for document in closest_documents:
            print(f"\033[95m%4s:␣" % str(i),  end="\033[0m")
            print_text = format_print_text(document[0], i)
            print(f"%-s␣%.4f"% (print_text, document[1]))
            i+=1

        selected_document = input("\033[92mEnter␣the␣number␣of␣the␣document␣
            to␣view\033[0m␣(n/new␣for␣new␣prompt):␣")
        if selected_document == "q" or selected_document == "quit" or
            selected_document == "exit":
            exit(0)
        if selected_document == "n" or selected_document == "new":
            break

        try:
            selected_document = int(selected_document)
            if selected_document < 0 or selected_document >= len(
                closest_documents):
                print("Invalid␣document␣number")
                continue
        except:
            print("Invalid␣document␣number")
            continue

        print(closest_documents[selected_document][0])
        temp_file_names, temp_data = gm.get_data(document_folder, data_limit
            )

        index = np.nonzero(np.array(temp_file_names) == closest_documents[
            selected_document][0])[0][0]
        print("-"*os.get_terminal_size().columns)
        print(temp_data[index])
        print("-"*os.get_terminal_size().columns)

def add_new_file_name(u_k, is_k, v_k, file_names, word_map, word_list,
    new_file_name, new_data):
    # Since we search for the closest document inside the V_k matrix, based
        on the altered query vector, we can transform a new document into a
        "query vector" and add it to the V_k matrix
    debug.log("Adding␣new␣document:␣" + new_file_name)
    new_document_vector = build_query_vector(new_data, word_map, word_list)
    altered_document_vector = np.dot(np.dot(new_document_vector.T, u_k),
        is_k)
    v_k = np.hstack((v_k, np.zeros((v_k.shape[0], 1), dtype=v_k.dtype)))
    v_k[:,-1] = altered_document_vector
    file_names.append(new_file_name)


    return u_k, is_k, v_k, file_names

def add_file_words(u_k, s_k, v_k, file_names, word_map, word_list, new_data)
:
    # Since documents are stored as columns in the V_k matrix, we can do an
        inverse transformation to add new words to the U_k matrix
    debug.log("Adding␣new␣words")
    data_split = new_data.split()
    new_words = set()
    for word in data_split:
        if word not in word_map:
            new_words.add(word)

    debug.log("New␣words:␣" + str(new_words))
    for word in new_words: # Add every new word we found in the data to U_k
        one by one
        debug.log("Adding␣word:␣" + word)
        word_map[word] = len(word_list)
        word_list.append(word)
        count = 0
        for j in range(len(data_split)): # Check how many times the new word
             appears in the new document data
            if word == data_split[j]:
                count += 1
        word_query_vector = np.zeros(len(file_names))
        debug.log("Length␣of␣word_query_vector:␣" + str(len(
            word_query_vector)))
        debug.log("Length␣of␣file␣names:␣" + str(len(file_names)))
        debug.log("shape␣of␣v_k␣" + str(v_k.shape))
        word_query_vector[-1] = count
        altered_word_query_vector = np.dot(np.dot(word_query_vector, v_k.T),
             s_k)
        u_k = np.vstack((u_k, np.zeros((1, u_k.shape[1]), dtype=u_k.dtype)))
        u_k[-1, :] = altered_word_query_vector

    return u_k, s_k, v_k, word_map, word_list

def add_new_documents(u_k, s_k, v_k, file_names, word_map, word_list):
    debug.log("Adding␣new␣documents")
    is_k = s_k_inverse(s_k)
    new_file_names, new_data = gm.get_new_data(document_folder, file_names,
        data_limit+new_data_limit)
    for i in range(len(new_file_names)):
        # Add new document
        u_k, is_k, v_k, file_names = add_new_file_name(u_k, is_k, v_k,
            file_names, word_map, word_list, new_file_names[i], new_data[i])

        # Add new word???
        u_k, s_k, v_k, word_map, word_list = add_file_words(u_k, s_k, v_k,
            file_names, word_map, word_list, new_data[i])

    return u_k, s_k, v_k, file_names, word_map, word_list

def setup_parser():
    global document_folder
    global k
    global cosinus_threadhold
    global modes
    global mode
    global save_files_folder
    global recompute
    global data_limit
    global new_data_limit
    global should_add_new_documents
    global testing
    parser = argparse.ArgumentParser(description='Find␣the␣closest␣document␣
        to␣a␣prompt')

    parser.add_argument('--folder', type=str, help='The␣folder␣containing␣
        the␣documents.␣Default␣is␣"' + document_folder + '"', default=
        document_folder)
    parser.add_argument('-o', '--online', help='Whether␣to␣use␣the␣online␣
        library␣of␣data.', action='store_true')
    parser.add_argument('-m', '--mode', type=str, help='The␣mode␣to␣run␣in.␣
        Options␣are␣' + str(modes) + '.␣Default␣is␣' + modes[0] + '.',
        default=modes[0])
    parser.add_argument('-k', '--k', type=int, help='The␣number␣of␣singular␣
        values␣to␣use.␣Default␣is␣' + str(k) + '.', default=k)
    parser.add_argument('-c', '--cosine', type=float, help='The␣cosine␣
        similarity␣threshold.␣Default␣is␣' + str(cosinus_threadhold) + '.',
        default=cosinus_threadhold)
    parser.add_argument('-d', '--debug', help='Print␣debug␣information.',
        action='store_true')
    parser.add_argument('--compute', help='Compute␣all␣files␣again.', action
        ='store_true')
    parser.add_argument('-l', '--limit', type=int, help='The␣max␣number␣of␣
        documents␣to␣use.␣Default␣is␣' + str(data_limit) + '.', default=
        data_limit)
    parser.add_argument('-a', '--add', help='Add␣all␣new␣documents␣to␣the␣
        database.', action='store_true')
    parser.add_argument('-al', '--addlimit', type=int, help='The␣max␣number␣
        of␣new␣documents␣to␣add.␣Default␣is␣unlimited.', default=1000000000)
    parser.add_argument('-t', '--test', help='Run␣the␣test␣suite.␣Always␣
        uses␣online␣data.␣Generates␣new␣tables.', action='store_true')
    args = parser.parse_args()

    is_online = args.online
    if args.test:
        is_online = True
    if is_online:
        document_folder = ""
    if args.folder and not is_online:
        document_folder = args.folder
    if args.k:
        k = args.k
    if args.cosine:
        cosinus_threadhold = args.cosine
    if args.mode:
        if args.mode not in modes:
            debug.log("Invalid␣mode.␣Options␣are␣" + str(modes))
            exit()
        mode = args.mode
    if args.debug:
        debug.set_debug_level(1)
    if args.compute:
        recompute = True
    if args.limit:
        data_limit = args.limit
    if args.addlimit:
        new_data_limit = args.addlimit
    if args.add:
        should_add_new_documents = True
    else:
        new_data_limit = 0
    if args.test:
        testing = True

    # Due to many different configurations, we save the data in different
        folders based on the significant parameters
    save_files_folder += mode + "_"
    save_files_folder += str(data_limit) + "_"
    if is_online:
        save_files_folder += "online"
    else:
        save_files_folder += document_folder
    if args.test:
        save_files_folder += "_test"


    debug.log("Folder:␣" + document_folder)
    debug.log("K:␣" + str(k))
    debug.log("Cosine:␣" + str(cosinus_threadhold))
    debug.log("Mode:␣" + args.mode)
    debug.log("Online:␣" + str(is_online))

def check_saved(): # Check if we have all the necessary precomputed data, to
     avoid recomputing
    global files_saved

    debug.log("Checking␣sava␣data")
    files_saved = True
    if not os.path.isdir(save_files_folder):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, u_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, s_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, v_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, file_names_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, word_map_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, word_list_file)):
        files_saved = False
    if not os.path.exists(os.path.join(save_files_folder, matrix_file)):
        files_saved = False

    if recompute: # If we want to force a recompute
        files_saved = False


    if not files_saved: # If an of the files are missing, delete all the
        data and start over
        debug.log("Data␣not␣saved")
        if os.path.isdir(save_files_folder):
            shutil.rmtree(save_files_folder)
        os.makedirs(save_files_folder)


    return files_saved

def run():
```

```
global k
global document_folder
global testing

files_saved = check_saved()
file_names, word_map, word_list, matrix = None, None, None, None

gm.testing = testing
debug.log("Getting␣data")
if not files_saved: # If we don't have the data saved, we need to
    compute it and save it
    optimize = False
    if mode == modes[1]:
        optimize = True

    debug.log("Computing␣data")
    file_names, word_map, word_list, matrix = gm.generate_matrix(
        document_folder, optimize, data_limit)
    np.save(os.path.join(save_files_folder, file_names_file), file_names
        )
    np.save(os.path.join(save_files_folder, word_map_file), word_map)
    np.save(os.path.join(save_files_folder, word_list_file), word_list)
    np.save(os.path.join(save_files_folder, matrix_file), matrix)
else:
    debug.log("Reading␣saved␣data")
    file_names = np.load(os.path.join(save_files_folder, file_names_file
        ), allow_pickle=True).tolist()
    word_map = np.load(os.path.join(save_files_folder, word_map_file),
        allow_pickle=True).item()
    word_list = np.load(os.path.join(save_files_folder, word_list_file),
         allow_pickle=True).tolist()
    matrix = np.load(os.path.join(save_files_folder, matrix_file))

u_k, s_k, v_k = svd(matrix, k, files_saved)
is_k = s_k_inverse(s_k)

if should_add_new_documents:
    u_k, s_k, v_k, file_names, word_map, word_list = add_new_documents(
        u_k, s_k, v_k, file_names, word_map, word_list)


if testing: # If we are running the test suite. Here we remove the file
    information from the training data, then feed the file "subjects" as
     the search prompt. We only ever do this on online files
    subjects = gm.get_subjects(data_limit+new_data_limit) # Get the
        subjects of the documents
    temp_file_names, temp_data = gm.get_data(document_folder, data_limit
        +new_data_limit) # Get the titles and data of the documents
    i = -1
    print("Testing")
    for subject in subjects:
        i+=1
        subject = subject.lower()
        debug.progress(i, len(subjects), without_debug=True)
        q = build_query_vector(subject, word_map, word_list)
        q_altered = np.dot(np.dot(q.T, u_k), is_k)
        closest_documents = find_closest_documents(q_altered, v_k,
            file_names)
        testing_analysis(closest_documents, subject, temp_file_names,
            temp_data)
    print("Score:␣" + str(testing_score))
    # print(testing_score)
    while not testing: # If we are not running the test suite, we can search
         for documents based on a prompt. The query vector is calculated
        based on given formulas
        prompt = input("\033[92mEnter␣a␣prompt\033[0m␣(q/quit/exit␣to␣quit):
            ␣").lower().strip()
        if prompt == "q" or prompt == "quit" or prompt == "exit":
            break
        q = build_query_vector(prompt, word_map, word_list)
        q_altered = np.dot(np.dot(q.T, u_k), is_k)
        closest_documents = find_closest_documents(q_altered, v_k,
            file_names)
        analyze_results(closest_documents)

def main():
    setup_parser() # Arguments are parsed here

    run() # The main function is run here

if __name__ == "__main__":
    main()
```

7.2 generateMatrix.py

```
import docx
import numpy as np
import os
import debug
from sklearn.datasets import fetch_20newsgroups
import sys
import re

testing = False
subjects = []

def correct_data_for_testing(data):
    # Remove all non-alphabetic/number characters and make all characters
        lowercase
    for i in range(len(data)):
        data[i] = data[i].lower()
        data[i] = re.sub(’[\W_]+’, ’␣’, data[i])

    return data


def get_new_data(folder, prev_file_names, data_limit):
    debug.log("Getting␣new␣data")
    all_file_names, all_data = get_data(folder, data_limit)

    new_data = []
    new_file_names = []

    for i in range(len(all_file_names)):
        if all_file_names[i] not in prev_file_names:
            new_data.append(all_data[i])
            new_file_names.append(all_file_names[i])

    if testing:
        alter_data_for_testing(new_data)
    correct_data_for_testing(new_data)

    debug.log("New␣data␣found:␣" + str(len(new_data)))
    return new_file_names, new_data

def get_subject_from_document(data):
    # When testing, we need the "subject" of the testing data. This is the
        string after "Subject:"
    split = data.split("\n")
    for j in range(0, len(split)):
        split_line = split[j].split()
        if len(split_line) == 0:
            print(data)
            print("No␣subject␣found␣-␣empty␣line")
            exit(0)
        if split_line[0] == "Subject:":
            subject_string = split[j].split("Subject:")[1].split("Re:")[-1].
                strip()
            return subject_string
    else:
        print(data)
        print("No␣subject␣found␣-␣no␣lines")
        print("Data:" + str(i))
        exit(0)
```

```
def get_subjects(data_limit):
    data = get_data("", data_limit)[1]

    debug.log("Getting␣subjects")
    for i in range(len(data)):
        subject_string = get_subject_from_document(data[i])
        subjects.append(subject_string)

    return subjects

def alter_data_for_testing(data):
    global testing
    global subjects
    debug.log("Altering␣data␣for␣testing␣(removing␣metadata␣and␣subjects)")

    for i in range(len(data)):
        split = data[i].split("Lines:")
        data[i] = ""
        for j in range(1, len(split)):
            data[i] += split[j] + "␣"

def read_docx_file(filename):
    doc = docx.Document(filename)
    full_text = []
    for para in doc.paragraphs:
        full_text.append(para.text)
    return '\n'.join(full_text)

def read_file(filename):
    return read_docx_file(filename)

def get_data(folder, data_limit): # Get the data from the chosen files. If
    no folder is given, we assume that the online library data is wanted
    debug.log("Getting␣data")
    if folder == "": # Use the online files
        newsgroups_train = fetch_20newsgroups(subset='train')
        if data_limit>newsgroups_train.filenames.shape[0]:
            debug.log("No␣data␣limit")
            data_limit = newsgroups_train.filenames.shape[0]
        file_names = newsgroups_train.filenames[:data_limit]
        # for i in range(len(file_names)):
        #     file_names[i] = get_subject_from_document(newsgroups_train.
            data[i])
        return file_names, newsgroups_train.data[:data_limit]

    # Else, use the files in the given folder. This code can be expanded to
        include other file types
    file_names = os.listdir(folder)
    titles = []
    data = []
    for file_name in file_names:
        full_text = read_file(os.path.join(folder, file_name))
        data.append(full_text)
        titles.append(file_name)
    debug.log("Data␣gotten")
    return titles, data

def create_frequency_matrix(data): # data is a list of strings. Those
    strings are the content of the documents
    # Here we create the matrix, which counts how many times each word
        appears in each document. Since we are only storing integers, we do
        not need the document titles, as the data is already stored in the
        same order as the titles
    if testing:
        alter_data_for_testing(data)
    correct_data_for_testing(data)

    debug.log("Creating␣frequency␣matrix")
    column_count = len(data)
    word_set = set() # Set of all words
    word_map = {} # Map of word to index

    # Create a set of all words (no duplicates)
    debug.log("Creating␣word␣set")
    for i in range(column_count):
        full_text = data[i]
        words = full_text.split()
        for word in words:
            word_set.add(word)

    # Create a list of all words and sort them
    word_list = list(word_set)
    # word_list.sort()

    # Create empty matrix with rows = number of words and columns = number
        of documents
    row_count = len(word_list)
    matrix = np.zeros((row_count, column_count))

    # Create a hashmap of word to index, so we can quickly find the index of
         a word
    for i in range(row_count):
        word_map[word_list[i]] = i

    # Fill the matrix with the count of each word in each document
    for j in range(column_count):
        full_text = data[j]
        words = full_text.split()
        for word in words:
            matrix[word_map[word], j] += 1

    debug.log("Matrix␣created")
    return word_map, word_list, matrix

def matrix_to_file(matrix, file_name):
    # For debugging purposes, write the matrix to a file
    f = open(file_name, 'w' )
    f.write(np.array2string(matrix, threshold=np.inf, max_line_width=np.inf)
        )
    f.close()

def create_complex_matrix(matrix):
    # Here we transform our frequency matrix into the weighted version
        calculated with given formulas
    debug.log("Creating␣complex␣matrix")

    n = matrix.shape[1] # Number of documents

    debug.log("Number␣of␣words:␣" + str(matrix.shape[0]))

    debug.log("Number␣of␣documents:␣" + str(n))

    debug.log("Generating␣percentage␣matrix")
    percentageMatrix = matrix/matrix.sum(axis=1)[:,None]
    debug.log("Percentage␣matrix␣generated")
    percentageMatrix = percentageMatrix * np.log(percentageMatrix)
    percentageMatrix = np.nan_to_num(percentageMatrix)
    debug.log("Percentage␣matrix␣log␣multiplied")
    percentageMatrix = percentageMatrix / np.log(n)
    debug.log("Percentage␣matrix␣divided␣by␣log(n)")
    percentageMatrix = percentageMatrix.sum(axis=1)
    debug.log("Percentage␣matrix␣summed")
    percentageMatrix = 1 - percentageMatrix # Matrika globalnih mer

    matrix = np.log(matrix + 1)
    matrix = matrix * percentageMatrix[:,None]

    debug.log("Complex␣matrix␣created")
    return matrix

def generate_matrix(folder = "", optimize = False, data_limit = 100000000):
    debug.log("Generating␣matrix")

    # file_names = os.listdir(folder)
    # Matrix of the number of times a word appears in each document
    titles, data = get_data(folder, data_limit)
    word_map, word_list, matrix = create_frequency_matrix(data)
    # Matrix based on point 4, local and global importance optimization.
        Uncomment for optimization
    if optimize:
        matrix = create_complex_matrix(matrix)
    return titles, word_map, word_list, matrix
```

7.3 debug.py

```python
import sys

debug_level = 0

def log(message):
    if debug_level == 0:
        return
    print(message)

def set_debug_level(level):
    global debug_level
    debug_level = level

def progress(current, total, without_debug=False):
    if debug_level == 0 and not without_debug:
        return

    if total > 100:
        if current % int(total/100) != 0:
            return

    sys.stdout.write('\r')
    percentage = 100 * current / total
    num_symbols = int(percentage/100*20)
    sys.stdout.write("[\033[92m%-20s\033[0m]␣%d%%␣" % ('='*(num_symbols),
        int(percentage)))
    sys.stdout.flush()
```