Recommendation Application

Rosa de Freitas(6777783) & Jelle Tuik(6617697)

1 Introduction

This report discusses the design process of a minimalistic recommendation application implemented in Python. In this system a user gives a document as input, and the system recommends similar documents. The document a user gives as input is called a document query, and contains at least 50 words. The system uses a collection of ten plain text documents containing around 150-400 words to make its recommendations. These documents will be discussed more detailed in section 2 of this report. To retrieve the right documents and have them ranked on their relevance a vector is created consisting of tf-idf values for each document and document-query. The vectors of the query are then matched against the vectors of the documents making use of the cosine distance measure. The top 5 documents are the output ranked by distance. To make the system easier to use a web-based GUI using Flask will be made. The GUI will contain multiple ways to enter a document-query for input, the user can paste the text of the document, upload a .txt file or select a document from the collection. At the bottom is a button to calculate the recommendation. The GUI will also show the five best ranked documents, their file names, corresponding cosine distances, level of recommendation supported with a color scheme and a preview of the content of each document. How all of this is implemented will be further described in section 3. After everything is implemented an evaluation will be done by looking at the precision and recall of the system. This will be discussed in more detail in section 4. At the end of this report, in section 5, possible improvements will be discussed.

2 Dataset Description

The dataset consists of ten plain text documents, with a word amount varying between 185 and 407 and an average of 284 words. Below in table 1 each file and their total amount of words is shown.

***Table 1**: Total Amount of Words per Document*

Document	Total amount of words
Document1.txt	223
Document2.txt	224
Document3.txt	407
Document4.txt	228
Document5.txt	313
Document6.txt	221
Document7.txt	338
Document8.txt	185
Document9.txt	392
Document10.txt	308

All of the documents are news articles from the NOS website and written in Dutch. The topics of the documents are the multinationals Apple and Shell. Four out of the ten documents are about Apple, another four are about Shell, and the remaining two documents are about both Apple and Shell. The articles of the website were put in .txt files to enable these to be read in the recommendation application. Therefore these documents do not have any parts which are not really part of the document.

To make sure the dataset is usable in the recommendation application four pre-processing steps need to be performed. Firstly, a lexical analysis of the documents is done. This means that all uppercase letters are turned into lowercase letters, all interpunction is removed and that all stop words are removed. To turn uppercase letters into lowercase letters the .lower() statement is used. The interpunctions are removed by defining the interpunctions as follows: *punctuation= '"!()-[]{};:'"\, <>./?@+#$%^&*_~'"*, and the code in figure 1.

```
for lines in doc_open:
    for words in lines.split():
        remove_capitals = words.lower()
        non_capital_list.append(remove_capitals)
for punc in non_capital_list:
    for character in punc.split():
        remove_punctuation = character.strip(punctuation)
        cleaned_list.append(remove_punctuation)
```

Figure 1 *code of removing punctuation*

When doing a lexical analysis decisions need to be made about particular words and numbers. Some numbers in documents can be relevant for a query, for example numbers representing a year. Another thing to consider are the importance of plural words with a " 's " (e.g. extra's). In the code shown in figure 2 stemming is done. Words like extra's, will change to extra.

```
for word in revised_list:
    for letter in word.split():
        short_word = letter.translate({ord(i): None for i in punctuation})
        end_list.append(short_word)
```

Figure 2

In Dutch there are words connected with a -, like Noord-Holland. This interpunction is removed during the removal of interpunction because the words will keep their meaning after the interpunction is removed and they can still be found as a query result.
The stop words are removed with the help of the nltk library. Below in figure 2 and 3 the use of this library is shown.

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stops = set(stopwords.words('dutch'))
```

Figure 3 *import of nltk library*

```
for word in cleaned_list:
    if word not in stops:
        revised_list.append(word)
```

Figure 4 *use of nltk library in code*

The second part of the pre-processing is calculating the term frequencies for each term. The term frequencies are put in a dictionary, which is put in another dictionary. These frequencies are put in a term frequency matrix. The matrix has a structure of a list in lists. Rows correspond to the terms, and the columns to the documents. The first row of the matrix consists of the word 'term' and the names of the documents. Then all of the terms are processed, each with their own row in the matrix which shows the term frequency in each document. After this is done the term frequency matrix is made into a term weight matrix. For this the number of nonzero frequencies is counted. Then the N value needs to be calculated which is the total number of documents in the collection and the document-query. Then the

inverse document frequency can be calculated with log2(N/df). All the term frequencies are multiplied by this idf and stored in a matrix. This finished matrix will be the input for the recommendation engine.

3 Method and Implementation

Below a high-level diagram can be found which shows how the system operates in figure 5. The user gives the input in the form of a document-query to the recommendation engine. The recommendation engine then pre-processes the document-query. The query is then compared to the documents in the dataset. The documents with the highest similarity will be ranked and passed back to the recommendation engine, which shows the results to the user. Below more detail will be provided about how the similarity between the document-query and dataset are calculated.

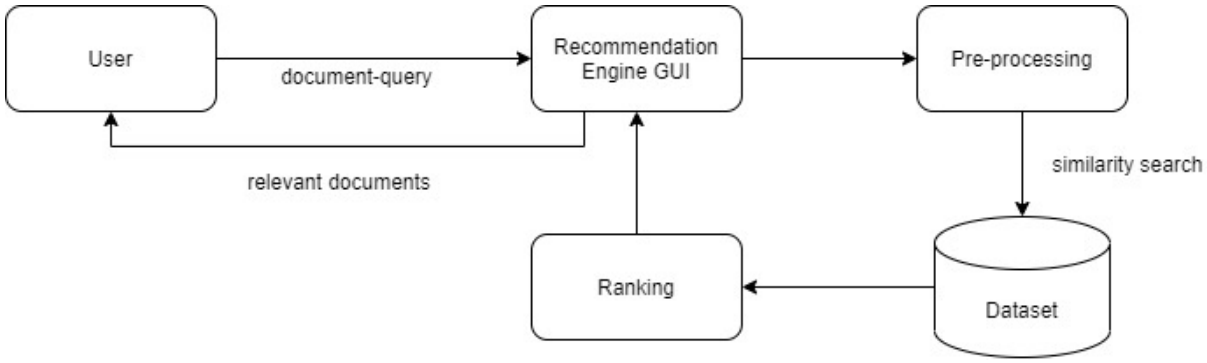

Figure 5 *high-level diagram of proposed recommendation engine*

Once all pre-processing steps of section 2 are finished, the recommendation engine can be implemented. The term weight matrix is read and the document vector lengths are calculated, for both the documents and the document-query. The following formula was used for calculating the vector lengths:

$$length(doc) = \sqrt[2]{tw_1^2 + tw_2^2 + \cdots + tw_n^2}$$

Here *tw* stands for term weight. The results, which are the vector lengths, are stored in a matrix. After this is done the cosine similarities are calculated for each document. This is calculated with the following formula:

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}$$

In the formula *d* stands for the document and *V* for the query vector. In the top part the inner product between document and document-query vectors is calculated first. The vectors are multiplied pairwise. Then for each document all the weights corresponding to terms in the document-query are added up. This results in list with the accumulated scores per document. After this the accumulated weights are divided by the product of the document vector and document-query vector lengths. Then the output will be ranked, showing the most relevant documents, their similarity score and their scale.

4 Evaluation

The code has been evaluated a lot during the coding process, by having intermediate print statements to check if the right output is given. This was done with each step before moving on to the next step. To evaluate the finished system four different document-queries can be

used. Two queries are documents consisting of different sentences of a few of the documents in the dataset. The third query consists of dummy (lorem ipsum) text. The fourth query in an article that is about Apple and Shell but does not have many other overlap with the texts in the dataset. These queries include outputs of:

- Two that performs really well (document-query 1 en 2)
- One that performs OK-ish (document-query 4)
- One that returns (almost) no relevant documents (document-query 3).

Unfortunately, due to the code not working as expected no clear output could be measured.

5 Discussion

The finished system did not meet all the requirements due to issues with fully understanding the code. The code could only process very specific/hard coded input, which resulted in a code that could not process the documents in the dataset. The code was not flexible enough. Also due to time crunch not everything could be implemented as intended. Besides this, in case of a fully functioning recommendation engine, an improvement would be the dataset. This could be extended to provide more variety in output documents and more useful output to the given input. The pre-processing could be improved by adding lemmatisation to make searching for the right words more efficient.

5.1 Contributions

The following table shows the contributions of each team member, rounded off to half hours.

activity	**Rosa de Freitas**	**Jelle Tuik**
Recommendation engine		
1. read tfidf matrix	2	4
2. calculate document vector length	0.5	3
3. querying	1	8
4. show ranked output	2	7
Preprocessing		
1. calculate term frequencies	2	4
2. create term frequency matrix	1,5	8
3. transform to tfidf matrix	3,5	3
GUI and integration		
1. create basic program	2	6
2. integrate recommendation engine	3,5	5
3. integrate preprocessing	0.5	nvt
4. document previews and final steps	1	nvt
Evaluation and documentation		
1. testing code	3	nvt
2. IR-evaluation	-	nvt
3. Documentation	6,5	nvt