Project Report - Search Engine

Achille Desreumaux, Nicolas Joué, Melvyn Provenzano

November 2024

1 Introduction

Since their emergence in the 1990s, search engines have significantly changed the way we access information. They have become essential tools for billions of users, enabling them to navigate the vast amount of data available online. In their early days, these engines worked on simple searches based on keywords. Over time, they have evolved to incorporate increasingly powerful algorithms, using advanced techniques such as semantic search and, more recently, artificial intelligence.

For this project, we had the opportunity to create our own search engine applied to a corpus of 2,000 Wikipedia articles in French. The aim was to learn how to design a system capable of finding the most relevant article for each query, by applying information retrieval techniques seen in class. This work enabled us to explore the various stages involved, including data preprocessing, the construction of vector models and the evaluation of the relevance of each article.

2 Problem definition

The objective of this project is to build a search engine, on 2000 french wikipedia articles, that would provide the most relevant article from the corpus for each query provided.

In order to build this search engine, we were provided with 2000 french wikipedia articles extracted as text files alongside a JSONL file called requetes.jsonl that will be used in the following sections to evaluate our implementation.

3 Methodology

We decided to pursue a vector approach using TF-IDF features as we believed it was the most suited method for our data. This approach consist in representing documents and queries as vectors. We can then find the most relevant document for each query by calculating the cosine similarity between the documents and each query. Our project's pipeline includes the preprocessing of the documents, the construction of the TF-IDF matrix using a TF-IDF Vectorizer, and the processing of queries. Figure 1 shows a diagram of our pipeline split into the processing of the documents and the queries.

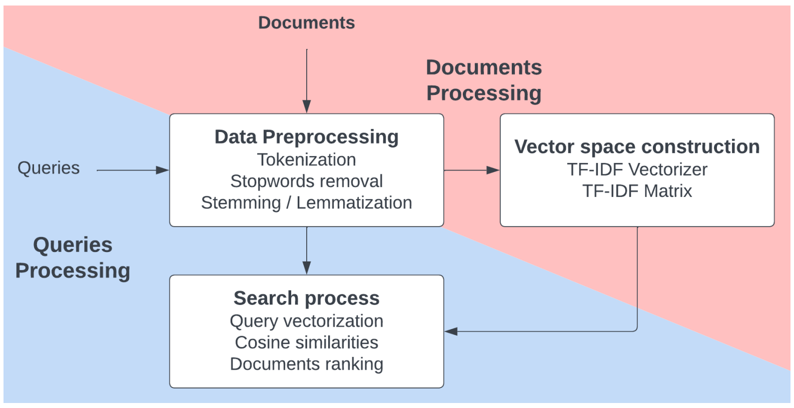

Figure 1: Diagram of the project's pipeline

Our first implementation relied on the TF-IDF Vectorizer from scikit-learn feature extraction library but we also implemented our own version to better understand the computations and calculations involved. The differences between these two implementations will be discussed in the results and discussions sections.

Our *search_engine.py* script uses the *argparse* library to offer different options when running it such as using our custom vectorizer or using stemming or lemmatization for the data preprocessing.

3.1 Data preprocessing

As described in the diagram of figure 1, the data preprocessing is done for each seperate document (french wikipedia article) in three steps.

First we put all characters to lowercase by using Python *.lower()* string method. Then we tokenize the string, getting rid of empty spaces and punctuation. To do so, we are using the Python regular expression library ***re*** using the pattern: r"\b\w[\w\-]+\b" and the method findall(). In the library, "\b" represents word boundaries, "\w" all alphanumerical characters, "\-" the hyphen such as in the word "porte-monnaie" and "[...]+" means we expect at least one occurrence of the content between brackets.

Then we can proceed to the removal of stopwords using the ***nltk*** library. *nltk* provides a list of 157 very common french words such as determiners (la, le, ce, ...), pronouns (je, j, tu, il, ...) as well as common verbs conjugations (est, suis, avez, ...). Removing stopwords can help improve the significance of TF-IDF scores as well as reduce the number of features in the matrix.

Finally, we can apply stemming to the remaining tokens using the *SnowballStemmer* from *nltk*. For lemmatization, the data preprocessing is different. Indeed, doing lemmatization requires more context, the word "avions" could either be a verb lemmatized to "avoir", or a noun lemmatized to "avion". This is what when using lemmatization, we use *spaCy* to perform both tokenization and lemmatization. The impact of using stemming or lemmatization will be further discussed in following sections.

3.2 TF-IDF Vectorizer

We explored two approaches to form the TF-IDF matrix. The first one involved using the TF-IDF Vectorizer from scikit-learn and the second one was based on our own implementation (from the file tfidf_vectorizer.py). The following explanations will cover our own implementation of the vectorizer that slightly differs from scikit-learn's. We will try to cover these differences.

The vectorizer needs to first be initialized before being used to create the matrix. Scikit-learn's implementation allows for many parameters to be entered, but we ended up using the default parameters and our implementation do not take parameters. In our implementation, we simply initialize the dictionnaries for the vocabulary, the IDF values and the list of documents.

Then we can use the vectorizer's *fit_ transform()* method to fit the documents and create the TF-IDF matrix at the same time. This method actually uses 2 other methods sequentially: *fit()* and *transform()*. The *fit()* method sets up the vocabulary, calculates the document frequency and subsequently the inverse document frequency (IDF). The *transform()* method initializes the matrix, count the number of terms in each document before computing the term frequency (TF) and enters the value in its position in the matrix. Finally, we perform a L2 normalization similarly to scikit-learn's implementation.

3.3 Queries processing

As illustrated by figure 1, queries are processed in the same way as documents are. We perform tokenization, stopwords removal and stemming or lemmatization. Then we use the vectorizer and its *transform()* methode to create a vector representing the query. It allows us to then calculate the cosine similarity between the query vector and the documents vectors in the TF-IDF matrix with the *cosine_ similarity()* method from scikit-learn.

Our script *search_ engine.py* offers two different modes for submitting queries. The *test* and *query* modes. The former allows to enter multiple queries at once via a JSONL file. The latter will launch a while loop that will enable the users to type in directly the queries and receive the most relevant article for each query.

In the *query* mode, we find the most relevant article using numpy's *argmax()* method and then we print out the article in the console. For *test* mode, we use numpy's *argsort()* method to obtain the list of articles sorted based on their similarity for each query. We will use this ranked list of articles to calculate the accuracy and the accuracy for finding the expected article in the five first results of a query. On top of it, there is a dedicated *test_ queries()* method to go through the JSONL file and print the right level of details of the results based on the indicated verbosity. The first level of verbosity only prints the queries that have not matched the expected file. The second level adds the 5 most relevant files found alongside their respective similarities. And the final level of verbosity also displays the correctly retrieved articles to the console.

4 Results

As stated previously, our search engine allows for different parameters to be tested. In this section, we will see the accuracy and the accuracy for the expected document to be in the 5 first results of the 100 queries provided in the *requetes.jsonl* file for different selection of parameters.

We will explore 4 alternatives : the use of scikit-learn's vectorizer or our own implementation with lemmatization or stemming. Scenario A is scikit-learn and lemmatization, B is scikit-learn and stemming, C is our vectorizer with lemmatization and D is our vectorizer with stemming.

	Accuracy	Top 5 documents Accuracy
Scenario A	82%	97%
Scenario B	85%	97%
Scenario C	81%	97%
Scenario D	85%	97%

Table 1: Accuracy and Accuracy of the expected document to be in the 5 first results of the 100 queries provided

As we can see from table 1, no matter the scenario chosen, the accuracy for the expected document to be in the five first results is 97%. The 3 queries that are not found in the 5 first results in all scenarios are: *"langue roumanie"*, *"metropolitain"* and *"Elizabeth Ière"*. For the first query, the article we are expected to find is *wiki_066072.txt* titled *"Lexique du roumain"*. For the second one, the article is *wiki_089929.txt* titled *"Métro de Paris"*. And for the third one, the article is *wiki_041649.txt* titled *"Élisabeth Ire (reine d' Angleterre)"*. For all three of these queries we can observe that words used in the query don't appear in the article because of a different spelling (Elizabeth instead of Élisabeth in the article) or the use of synonyms or related words (respectively *metropolitain* and *roumanie* instead of *métro* and *roumain* that do appear). This shows us that our search engine is sensitive to the exact words used and their spelling.

For the queries that get the expected document in the 5 most relevant documents but not at first place, one of the reason is that the words contained in the query appear more often in other articles. We can take the example of the two queries for the article *wiki_049182.txt* titled *"Bataille de France"* contains 7 times the word *"bataille"* or *"batailles"* and 8 times the word *"France"*. The scenarios using Lemmatization have the article *wiki_112022.txt* titled *"Bataille des Trois Rois"* ranked first or second, in which the term *"bataille"* appears 10 times even if the term *"France"* is absent. The scenarios that use stemming both have the article *wiki_066109.txt* titled *"Soulèvement national slovaque"* ranked first, in which the term *"bataille"* appears 24 times and for which the term *"France"* is also absent. And surprisingly, this second article is not in the 5 most relevant articles for the scenarios using lemmatization. This analysis is limited as the count of words is not directly equal to the TF-IDF scores but it still helps understand how the search engine can misrepresent the relevance of an article for queries that uses frequently used words in the corpus.

Talking about differences between the use of lemmatization and the use of stemming for the preprocessing step, we observed that the former were not able to find the right article for queries: *"24 heures du mans"*, and *"navette columbia"* when the latter did. The use of lemmatization or stemming may have slightly affected the distribution of terms in the documents and therefor the TF-IDF scores. For example, we saw earlier that many common verbs conjugations such as "est" or "avez" were present in the stopwords list. However their lemmatized form, respectively "être" and "avoir" is not. There might be more examples of stopwords not being removed due to the lemmatization. There might also be cases of words seemingly becoming more frequent due to the use of stemming. We can take the example of the noun "menace" and the verb "menacer" which both become "menac" with stemming whereas they would have remained in their original form with lemmatization.

We can also address the differences between our vectorizer and scikit-learn's. In the scenarios using lemmatization for the preprocessing step, the one using our implementation of the vectorizer is not able to find the correct most relevant article for the query *"définition énergie des mers"* whereas the one using scikit-learn's is able to. For the scenarios using stemming, despite having the same accuracy, our implementation succeeds on the query *"6e armee ottomane"* and fails on the query *"définition énergie des mers"*, while it is the opposite for theirs. We tried changing the formula for the IDF that we have seen in class to the formula they admitted using in their source code[1] but it did not allow us to find the same similarity scores. We noticed that their implementation of the vectorizer is much faster than ours. It might be possible that some of the optimizations used cause their TF-IDF scores to be slightly off. It is also possible that the error comes from our implementation and is subtle enough that it only slightly affects the scores computed.

5 Discussions

During this project, we were able to successfully design a NLP pipeline to create a document search engine. Depending on the parameters chosen, our search engine was able to identify the requested document as the most relevant one in 81% to 85% of the cases. Furthermore, when considering the 5 most relevant documents, this accuracy goes up to 97%. It is to be noted that the accuracies measured are very dependent on the corpus used as well as the queries. Using the same scripts but with different data could result in dissimilar performance.

We believe that the results we found are close to the best we could get using *bag of words* techniques. Indeed, except for the 3 outlier queries that we previously talked about, the difference in similarity for all other wrongfully predicted queries where very close to each other. Considering that most search engine will display multiple results to the user, this metric illustrates the good performance of the search engine.

Further improvements could include the use of semantic search or other semantic analysis techniques, sub-token tokenization (for example considering the token *"metropolitain"* as having *"metro-"* as a sub-token) or automatic spell checker on the queries. We could also probably use large language models (LLMs) through a Retrieval-Augmented Generation (RAG) model that would first summarize each document and store them on a database and later go through this database before providing the user with the most relevant article for their query.

[1]https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/feature_extraction/text.py