

Search Engine (Prácticas 1-7)



Đorđe Nikolić

Marzo 2020.

—

Sistemas de Recuperación de
Información

—

Pilar López Úbeda

Table of contents

Introduction	3
Diagrams	4
Main diagram	4
Preprocessing	5
Structures	5
Query handling	7
Search engine.....	8
Structure design	9
Development	11
Practice 1.....	11
Practice 2	11
Practice 3	12
Practice 4	12
Practice 5	13
Practice 6	14
Practice 7	15
Metadata.....	16
Application parameters	18
Application execution.....	19
Improvements.....	22
Web interface	22
API.....	22
Front-end.....	23
Configuration and execution.....	23
Web interface screenshots	24
Pseudo relevance feedback	26
Conclusion	27
Notes on improvements	27

Introduction

This document is part of the Search Engine project for the Sistemas de Recuperación de Información course at the University of Jaén in Spain. Contained within it is valuable information and explanations concerning the design decisions, implementation and general development of the project.

The development of this project was divided into 7 distinct parts, representing practices one through seven. The first three parts were directly related to the preprocessing of the input documents (html filtration, normalization, tokenization, stop word removal and stemming), while the fourth and fifth were dedicated to index building and other necessary structures. The last two were devoted to query processing and searching the index to get the wanted results.

This project was developed in the Spyder (Anaconda3) IDE using the Python programming language (version 3.7.4). The necessary packages to use the search engine are:

- nltk 3.4.5
- psutil 5.6.3
- BeautifulSoup4 4.8.0

Diagrams

- Main diagram

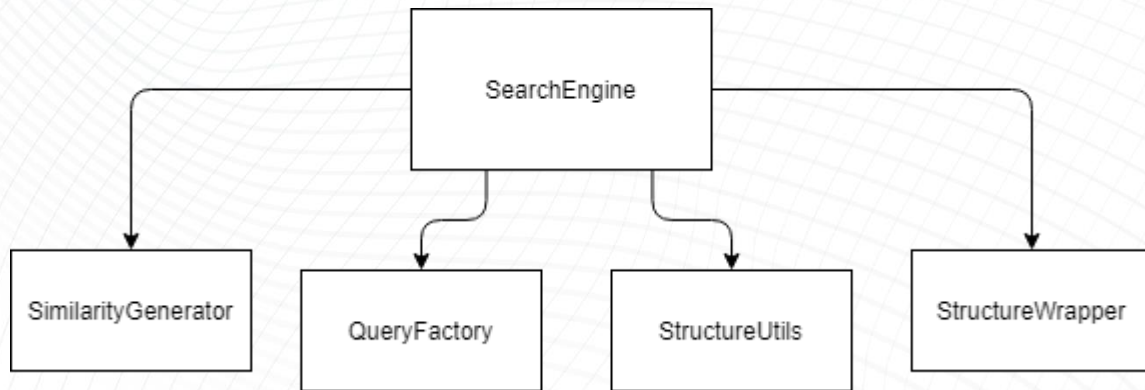


Figure 1.0 – Relations between the main classes of the project

- Preprocessing (Practices 1-3)

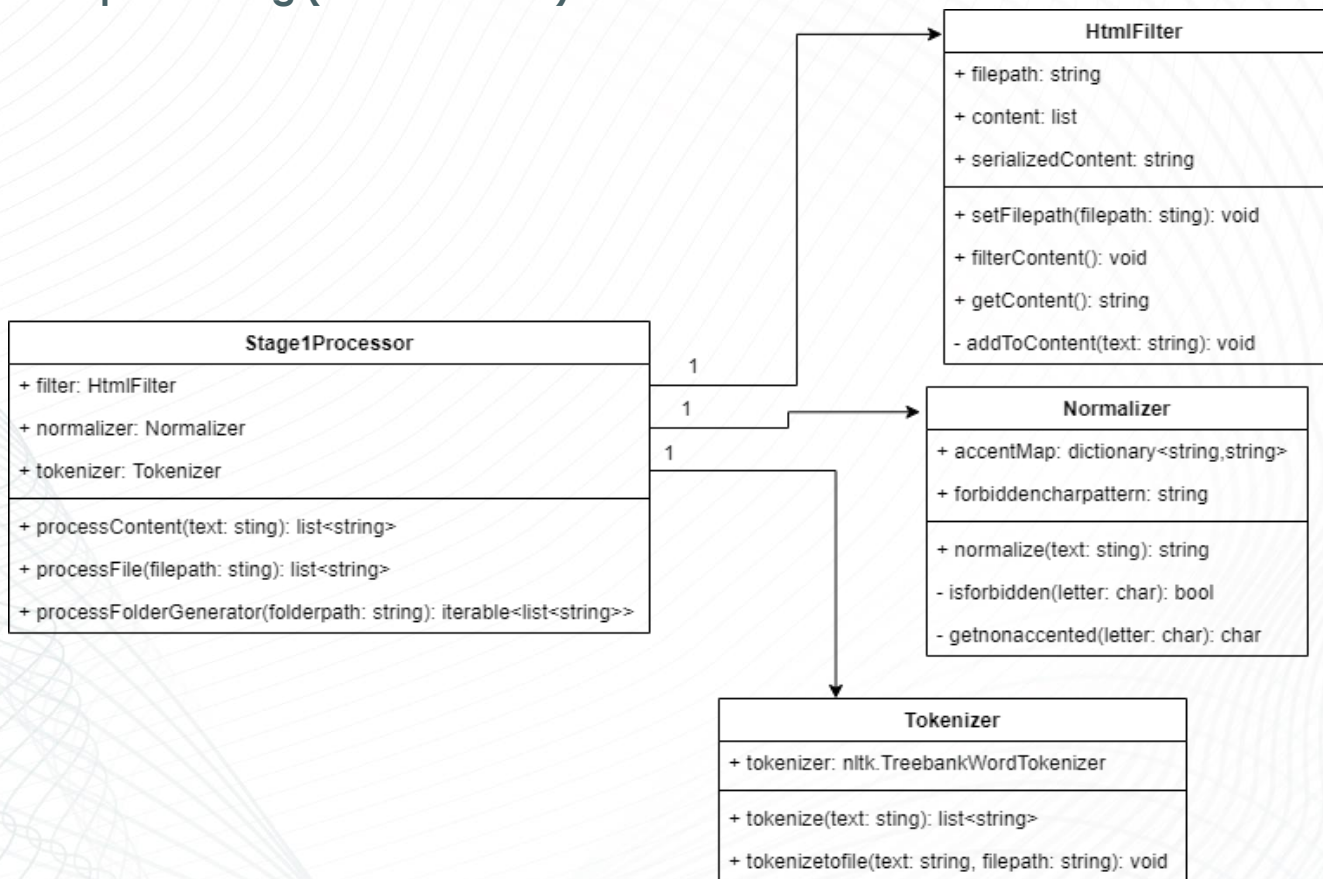


Figure 1.1 – Stage 1 preprocessing (text to words)

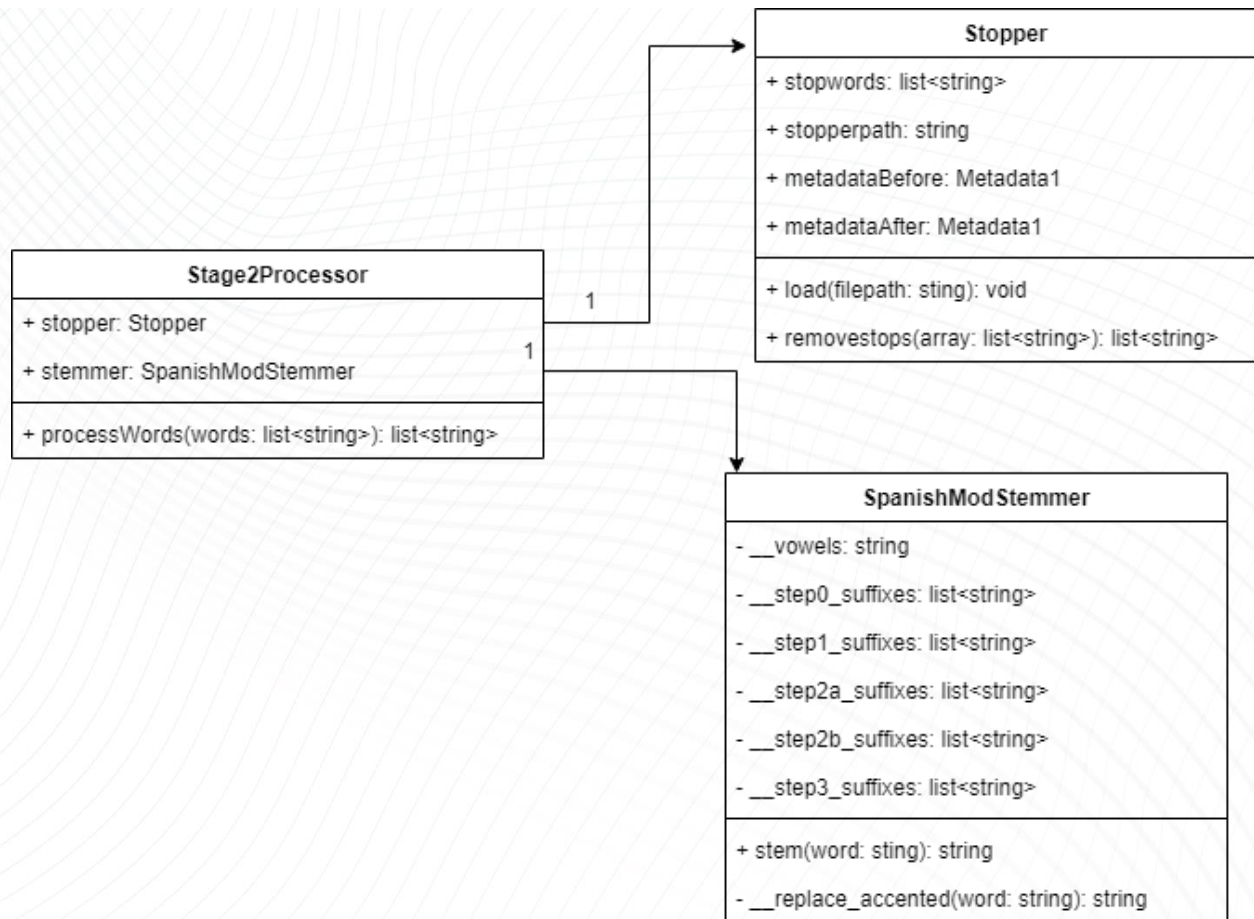


Figure 1.2 – Stage 2 preprocessing (word transformation)

- Structures (index, word references and file references) (Practices 4 and 5)

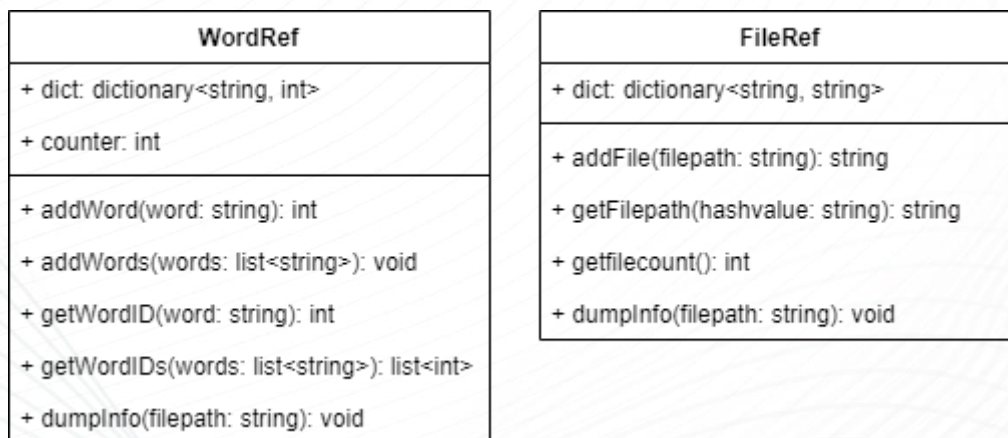


Figure 1.3 – Structures containing references

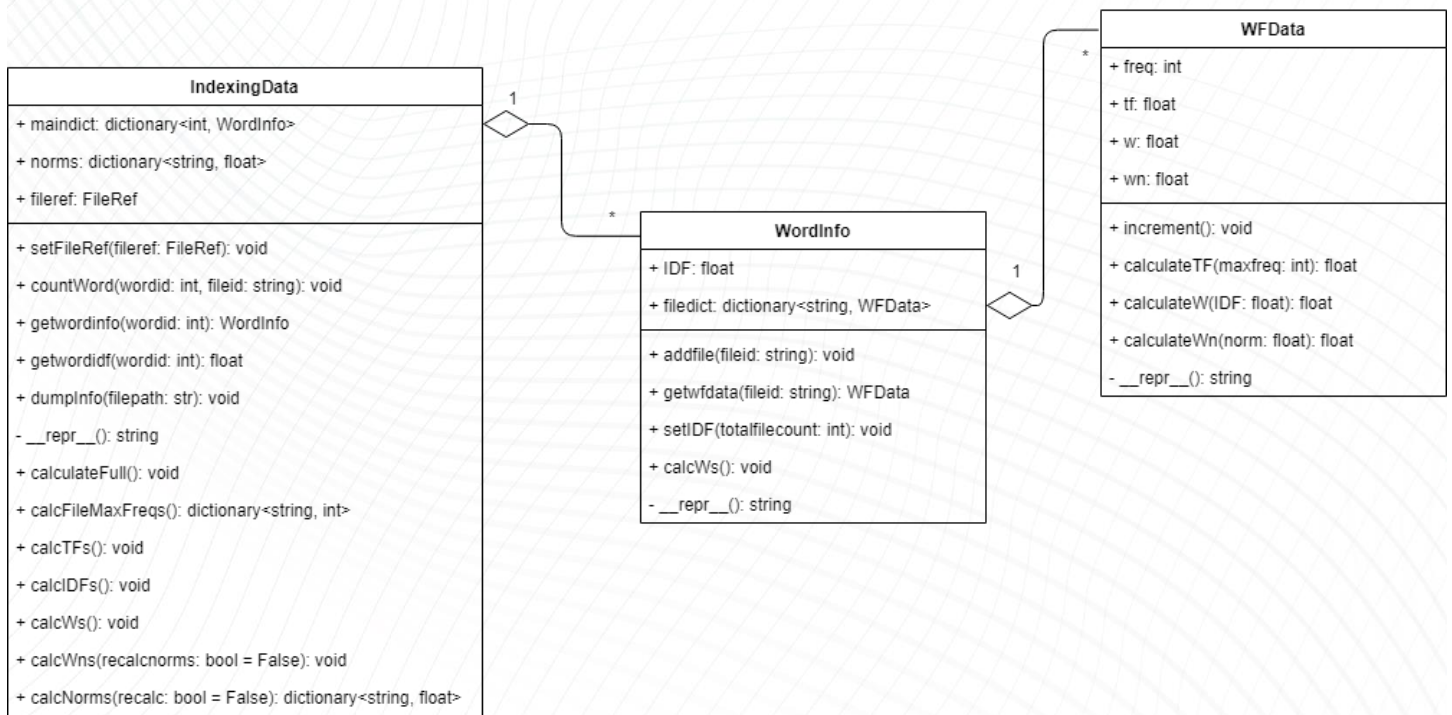


Figure 1.4 – Index structure(s)

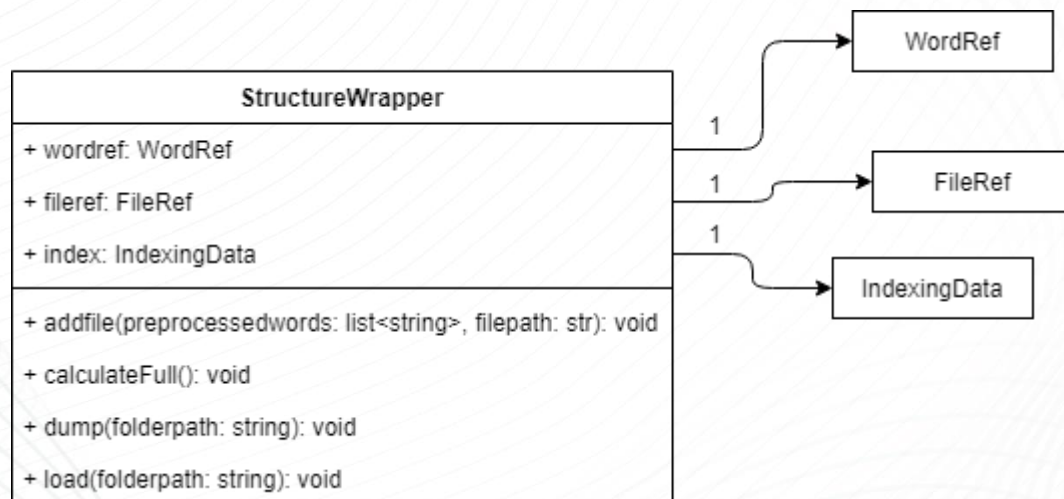


Figure 1.5 – Structure wrapper (use to handle lifetime grouping and loading/saving)

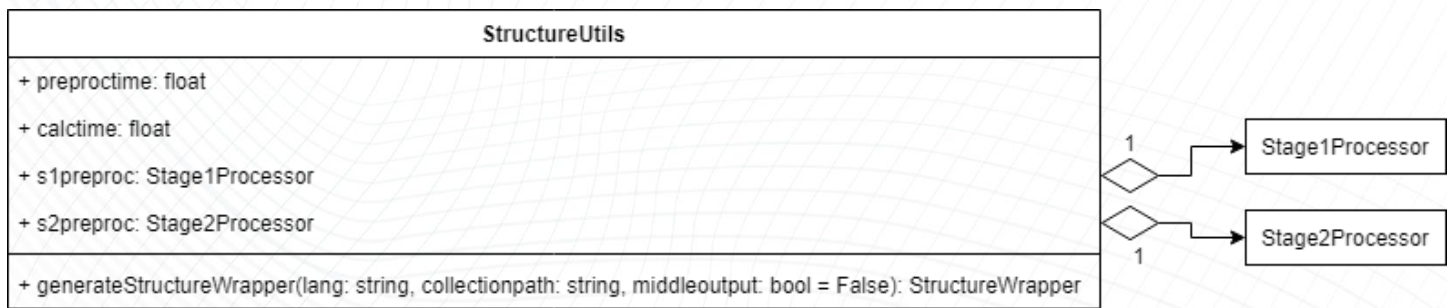


Figure 1.6 – Structure utility class (used to process all collection files and creates a *StructureWrapper*, along with all the structures contained within it)

- Query handling (Practices 6 and 7)

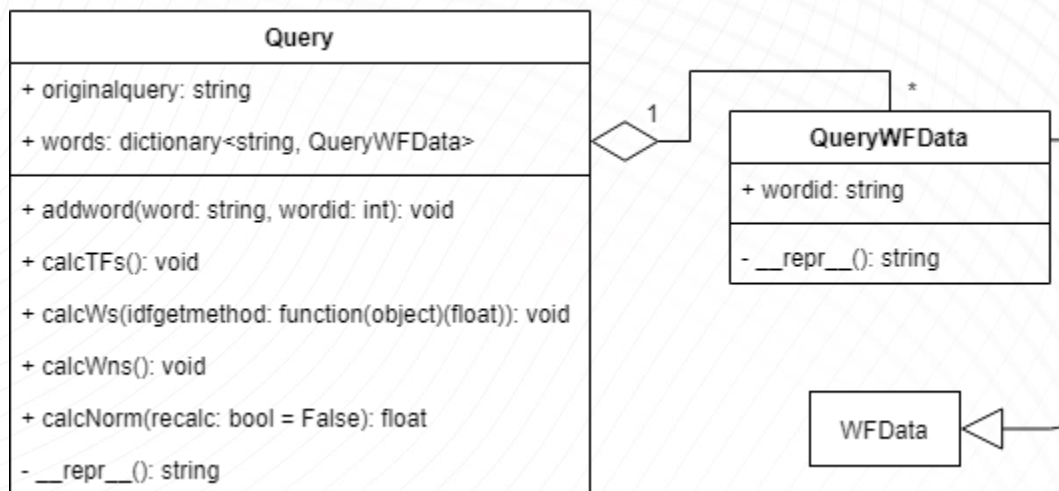


Figure 1.7 – Objects of these classes encapsulate all data needed before, during and after the processing of the input queries

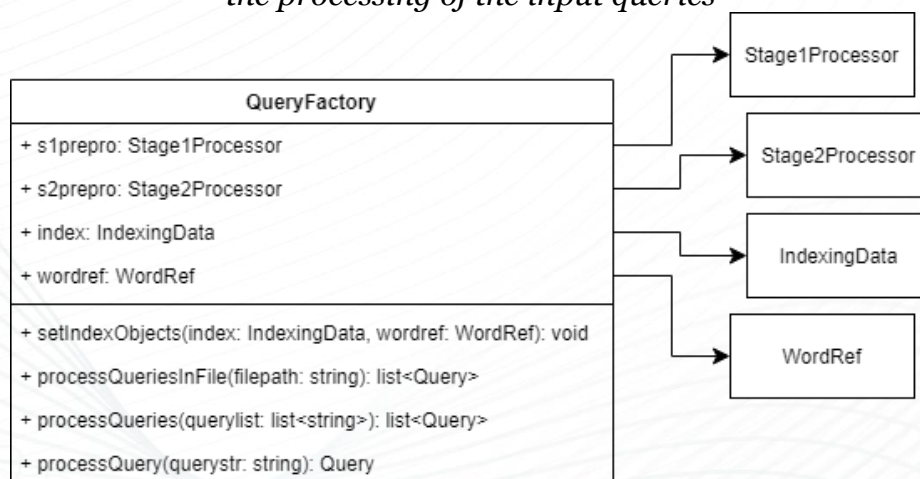


Figure 1.8 – This class is used to process input queries and create *Query* objects

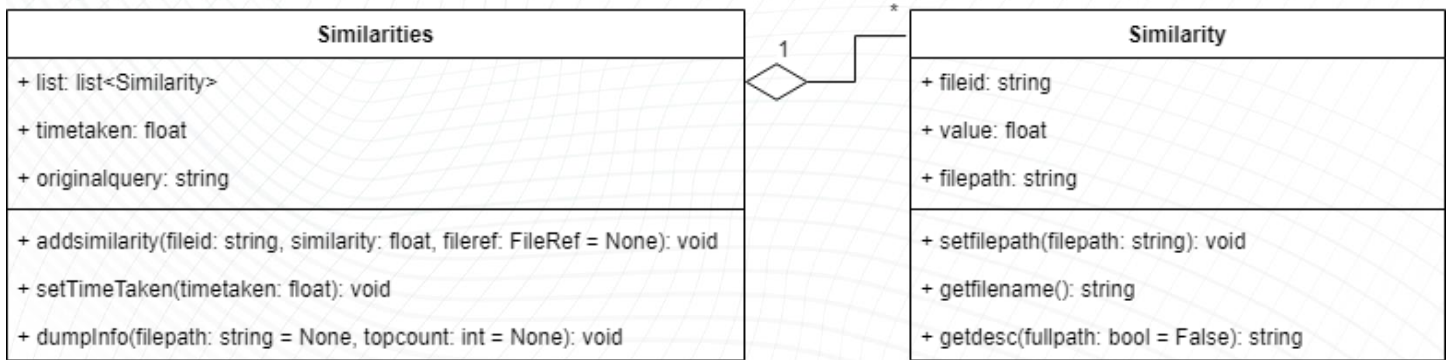


Figure 1.9 – Objects of these classes encapsulate result data from query search

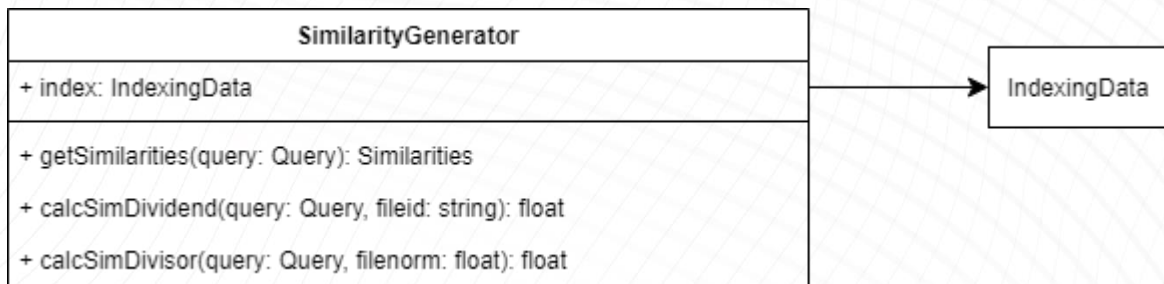


Figure 1.10 – This class is used to execute query search on a specified index and generate matching similarities

• Search engine (Practice 7)

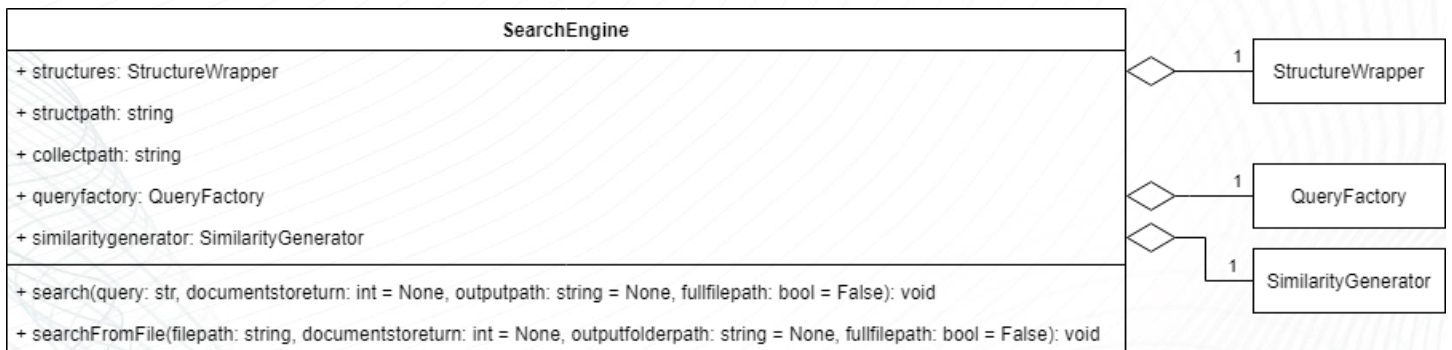


Figure 1.11 – Main class

Structure design

- **WordRef – word/token dictionary**

This structure is a simple dictionary, where the key is a string, and the value is an integer. The key is the word itself and the value is an ID by which we identify this word in all the other structures. The ID value is calculated as a simple auto-increment counter, located in the class.

We add words to this structure when they appear for the first time in one of the files from which we create the index. Because of this, all the words that are in this structure, appear in the index at least once. We use this to determine which words in an input query are relevant to the index and to find out their IDs.

- **FileRef – file dictionary**

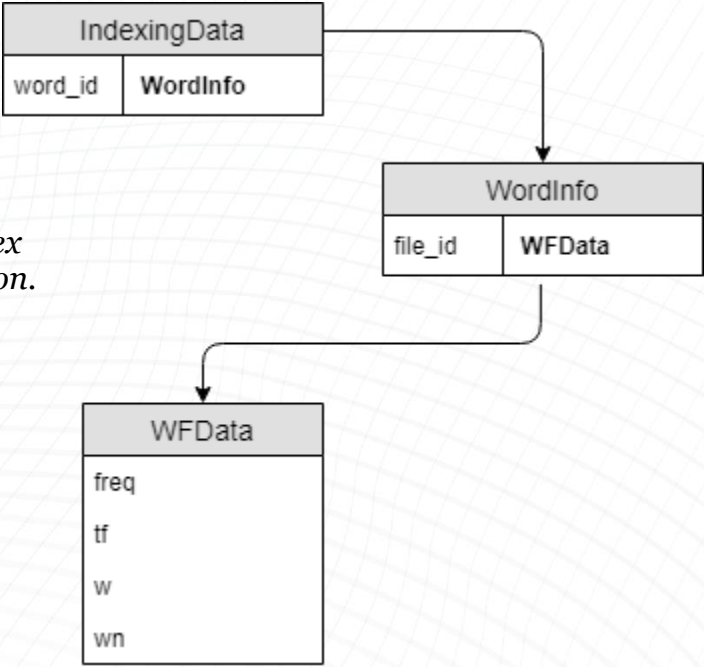
This structure is a dictionary, where both the key and the value are strings. The value is the full path to the file that we are adding, and the key is its hash value. The key is an ID by which we identify this file in the other structures. The hash value is calculated with the integrated Python `hash(str)` method.

We use this structure to find the full paths to the files that are determined to be relevant to the input query.

- **IndexingData – index**

The most important structure, the index. This structure is organized as a dictionary of dictionaries. With the parent dictionary having the signature of `<wordid, WordInfo>`, and the child dictionaries having the signatures of `<fileid, WFData>`. During development, the main question was whether to have the parent dictionary be accessed by the `wordid`, or the `fileid`. In the end, I decided to organize the parent dictionary by words, as this would allow direct access during query processing, and would allow the best performance. The trade-off is that index creation takes a little longer, as it's more difficult to calculate the maximum word frequencies per file and the norms used in normalized weight calculations.

Figure 2.0 – Index dictionary organization.



Development

• Practice 1

To start of this practice, and the whole project, we had to implement a module to clean up .html files and get only relevant content from them. I decided to use the [beautifulsoup4](#) package for this as it's well documented and widely supported.

I combine the text content of the `<title>` tag and the `<body>` tag along with all its descendants that are a `NavigableString`, while skipping comments and scripts.

Next, we had to implement a normalizing module, to process the resulting content. I go through the text, only once, changing all letters to lowercase, replacing any accented letter with its non-accented counterpart (á -> a, etc.), and removing any forbidden characters.

Only the following characters are allowed:

```
[ 'a' ... 'z', 'A' ... 'Z', ' ', '_', '-', '\n', '0' ... '9' ]
```

The rest are filtered out using a simple regular expression:

```
[^a-z0-9\s\_-\n]
```

After that, we were to create a tokenizing module. During development, I used a regular expression for this, but for the sake of consistency, I switched to the `nltk` implementation of the `TreebankWordTokenizer`.

Modules mentioned in this description:

- Practice1/htmlfilter.py
- Practice2/normalization_tokenization.py

• Practice 2

For the second practice, we had to implement a module that would be used to remove stop words from the resulting token list. On this [website](#), I found two .txt files containing Spanish stop words. I manually combined these two files into one file, contained in the Practice2 folder. During the

initialization of the module, this file is loaded into a dictionary, and duplicate words are removed.

Modules mentioned in this description:

- Practice2/stopper.py

• Practice 3

Our main task for this practice was to create or find a capable stemming module. During testing, I tried using the `SnowballStemmer` implementation from the `nltk` package, but I ran into some problems. Since we already removed all the accents from the text, this implementation of the stemmer had problems properly stemming the input tokens. So, after some research, I found out that the `nltk` package uses the *Apache 2.0 license*, which allows modification of the source code. I downloaded the source code for the `SnowballStemmer` and modified it slightly so that it worked with non-accented text. After some testing and comparisons, I concluded that it was adequate for this project.

Modules mentioned in this description:

- Practice3/stemmer.py

• Practice 4

The development of the modules in this practice took the most time since I had to think of the design thoroughly. First, we had to create two straightforward structures, to hold references to the words and files that appear in the index structure. These two structures are explained adequately in the **Structures** part of this document.

For the index structure, we had to make something that would properly hold word-file pairs and the frequency data. I decided on creating an inverted index structure, as it would later allow easier access. The first dictionary in this structure is accessed by the word ID, referenced from the previously mentioned word reference structure. The dictionary values are also dictionary objects that are accessed by the file ID, and hold an integer as values, these integers represent the word frequency in that file.

Modules mentioned in this description:

- Practice4/references.py
- Practice4/indexing.py

• Practice 5

Now, we had to expand on the previously created index structure, and allow storage and calculation of more data, including normalized weights. The child dictionary was changed to hold an object of the class `WFDData`, which encapsulates all needed data for that word-file pair. The child dictionary itself was also encapsulated into an object of the `WordInfo` class, along with data that was tied to the specific word, mainly **IDFs**.

The class holding the parent dictionary was also expanded to contain some caching data (file norms), and to handle all the calculations. The most difficult calculations to implement were the **tf** calculations and the file norm calculations, as they required multiple passes through the whole structure. During early development, these calculations took significant time, as I approached them by first going through the whole file reference structure and gathering data for each file individually. Later, I changed the algorithm so that it goes through the index structure itself, and maps the data found in a local dictionary, that I later go through again to process it. The execution time reduced by a factor of 10, as I went from a complexity of:

$$O(\text{num_files} * \text{total_word_count})$$

To a complexity of:

$$O(\text{average_files_per_word} * \text{total_word_count})$$

I also created a class to hold all three structures during their lifetimes and handle their saving and loading from files. I used the `pickle` package to handle serialization.

Modules mentioned in this description:

- `Practice5/indexing.py`
- `Practice5/structures.py`
- `Practice5/structures_serialization.py`

• Practice 6

For practice 6 we had to decide how to handle query processing from text and then how to find similarities in the index.

I created a module that generates **Query** objects from input text queries. These objects are designed to hold all data relevant to the words in the query and are also designed as an inverted index, but in this case, we only have one file, the query itself.

I have also created a module that uses the index and the given **Query** object to find similarities. This module also calculates the necessary data needed for this process, including normalized weights. **tfs** were calculated the same way as for the word-file pairs in the index. One problem here was how to handle words that do not appear in the index at all. **IDFs** for these words were calculated like this:

$$idf_i = \log_2 \frac{total_file_count + 1}{1}$$

Considering that these words are not located in the word reference, there is no data for them in the index, and they are ignored in the similarity calculations. I used this formula for the similarity calculations:

$$sim(d_j, q) = \frac{\sum_{i \in q} wn_{ij} * wn_{iq}}{\sqrt{\sum_{i \in j} wn_{ij}^2} * \sqrt{\sum_{i \in q} wn_{iq}^2}}$$

I have also developed a module for config parsing, it looks for two distinct lines in the loaded .txt file:

- `structures=path_to_folder_with_structures`
- `collection=path_to_collection_folder`

Modules mentioned in this description:

- `Practice6/queryhandling.py`
- `Practice7/similarity.py`
- `Practice6/config.py`

• Practice 7

For this last practice, I was mostly just writing classes and modules to properly organize the previous modules, while also refactoring some things in the modules created earlier. I expanded the module dedicated to similarity generation, so that it has powerful output capabilities, to satisfy the output and formatting requirements for this practice.

I also developed a main class, called `SearchEngine`. This class is what wraps every developed module up into a usable form. It is loaded using a path to the config file and some optional arguments. It parses the config file using a module developed in the previous practice. If it finds a valid path to a structure folder, it tries loading the structures. If it manages to load the structures, it initializes other necessary modules for query processing and similarity generation and completes the initialization. If structure loading is unsuccessful, or a valid path to the structure folder is not found, it tries to generate new structures from the supplied collection path. If this is also unsuccessful, it terminates with an error.

After the class has loaded it can process queries using two methods, `search(query)` and `searchFromFile(filepath)`. Both methods take some optional arguments related to the output location, display formatting, and the number of sorted documents to return. The first method takes the query string, processes it and matches it against the index, outputting the results to a file or to the standard output. The second one takes a file path to a query file, which contains one query string per line. It handles each of these the same way as the first method.

The class also contains a method to save the structures to a specified folder, `save(folderpath)`.

Modules mentioned in this description:

- `Practice7/similarity.py`
- `searchengine.py`
- `preprocessing.py`
- `structure_utils.py`

Metadata

I developed a separate module to measure and contain metadata for some of these practices. The mentioned module is `metadata.py`. All metadata results are contained in the `/metadata/` folder in the main package location.

• Practice 1 Metadata

```
Metadata1 (Practice1) stats:  
Files processed: 838  
Total collection tokens: 531736  
Average tokens per file: 634.5298329355609  
Time taken: 18.38745460016071
```

• Practice 2 Metadata

```
Metadata2 (Practice2 - before stop words removal) stats:  
Files processed: 838  
Total word count: 531736  
Average words per file: 634.5298329355609  
Minimum words in a file: 196  
Maximum words in a file: 1950  
Top 5 words by appearance:  
1. Word: de Count: 49758  
2. Word: la Count: 24982  
3. Word: el Count: 14685  
4. Word: en Count: 11892  
5. Word: y Count: 11781  
Time taken: 1.1713934000290465
```

```
Metadata2 (Practice2 - after stop words removal) stats:  
Files processed: 838  
Total word count: 306326  
Average words per file: 365.5441527446301  
Minimum words in a file: 156  
Maximum words in a file: 997  
Top 5 words by appearance:  
1. Word: jaen Count: 8423  
2. Word: diario Count: 5047  
3. Word: universidad Count: 4382  
4. Word: uja Count: 4052  
5. Word: 21 Count: 3437  
Time taken: 1.1713934000290465
```


• Practice 3 Metadata

```
Metadata2 (Practice3 - before stemming) stats:
Files processed: 838
Total word count: 306326
Average words per file: 365.5441527446301
Minimum words in a file: 156
Maximum words in a file: 997
Top 5 words by appearance:
1. Word: jaen Count: 8423
2. Word: diario Count: 5047
3. Word: universidad Count: 4382
4. Word: uja Count: 4052
5. Word: 21 Count: 3437
Time taken: 8.33084499996039
```

```
Metadata2 (Practice3 - after stemming) stats:
Files processed: 838
Total word count: 306326
Average words per file: 365.5441527446301
Minimum words in a file: 156
Maximum words in a file: 997
Top 5 words by appearance:
1. Word: jaen Count: 8423
2. Word: univers Count: 5866
3. Word: diar Count: 5062
4. Word: sit Count: 4195
5. Word: uja Count: 4052
Time taken: 8.33084499996039
```

• Practice 4 Metadata

```
Time taken: 0.7454186996474164
Structure info:
Type: <class 'Practice4.indexing.IndexingData'> Size: 11754432 b
System: Windows-10-10.0.18362-SP0
Processor: AMD64 Family 23 Model 24 Stepping 1, AuthenticAMD
Ram: 14 GB
```

• Practice 5 Metadata

```
Time taken: 0.6964298999992025
Structure info:
Type: <class 'Practice5.indexing.IndexingData'> Size: 55099498 b
System: Windows-10-10.0.18362-SP0
Processor: AMD64 Family 23 Model 24 Stepping 1, AuthenticAMD
Ram: 14 GB
```

• Practice 6 and 7 Metadata

The results of query processing itself are the metadata for these two practices and can be found in the *query_file_folder/Query Results/* folder after searchengine.py execution.

Application parameters

The main entry point for the application is the `searchengine.py` script. The script takes three command line arguments:

1. full file path to the config file
2. full file path to the query file
3. count of documents to return

These arguments are used to instantiate and configure an object of the class `SearchEngine`. Some additional parameters of the class are whether to generate middle files during preprocessing, which contain results from each stage of preprocessing and are stored in `collectionfolder/middle_output/`, and whether to display processing time. Both are set to `True` in this script.

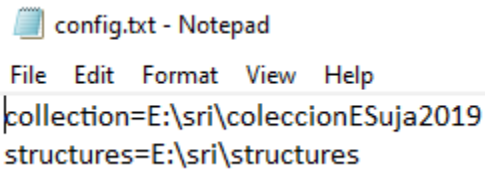
After the search engine is instantiated, the file located at the path supplied by the second argument is processed and its results are saved to files, located at `query_file_folder/Query Results/`.

After this is completed, the structures in the engine are saved to the specified structures folder path in the config file. If no folder path for structures is specified, this step is skipped, and the application terminates.

There are several other scripts included in the project (`main1to30.py`, `main4.py`, `main5.py`, `main6.py`). They were used during development and are there to display individual testing of the modules.

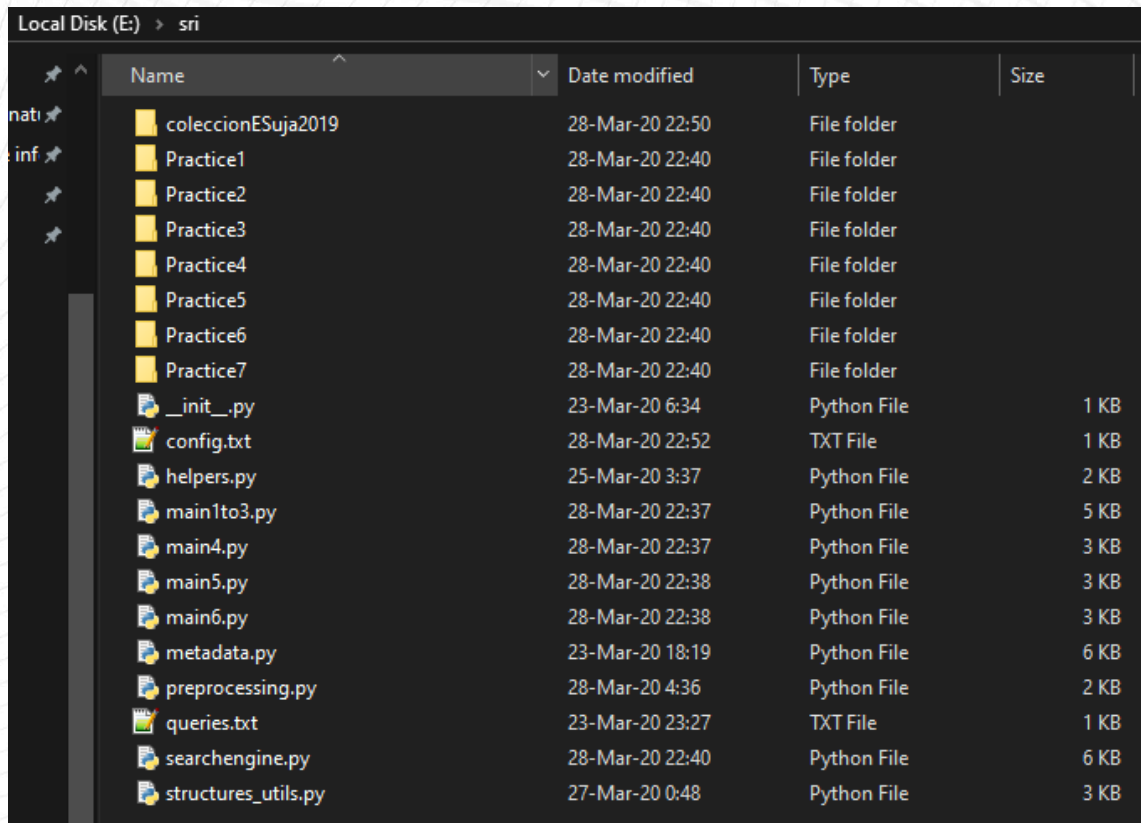
Application execution

The following are screenshots of the application and its environment:



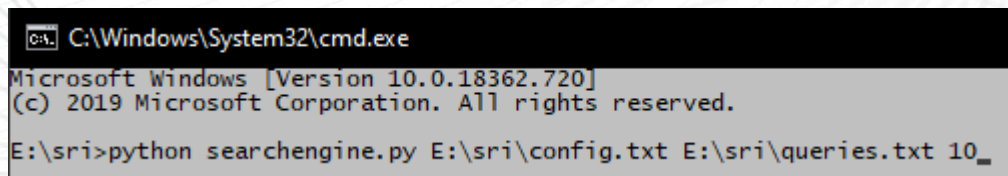
```
config.txt - Notepad
File Edit Format View Help
collection=E:\sri\coleccionESuja2019
structures=E:\sri\structures
```

Figure 3.0 – Config file



Name	Date modified	Type	Size
coleccionESuja2019	28-Mar-20 22:50	File folder	
Practice1	28-Mar-20 22:40	File folder	
Practice2	28-Mar-20 22:40	File folder	
Practice3	28-Mar-20 22:40	File folder	
Practice4	28-Mar-20 22:40	File folder	
Practice5	28-Mar-20 22:40	File folder	
Practice6	28-Mar-20 22:40	File folder	
Practice7	28-Mar-20 22:40	File folder	
__init__.py	23-Mar-20 6:34	Python File	1 KB
config.txt	28-Mar-20 22:52	TXT File	1 KB
helpers.py	25-Mar-20 3:37	Python File	2 KB
main1to3.py	28-Mar-20 22:37	Python File	5 KB
main4.py	28-Mar-20 22:37	Python File	3 KB
main5.py	28-Mar-20 22:38	Python File	3 KB
main6.py	28-Mar-20 22:38	Python File	3 KB
metadata.py	23-Mar-20 18:19	Python File	6 KB
preprocessing.py	28-Mar-20 4:36	Python File	2 KB
queries.txt	23-Mar-20 23:27	TXT File	1 KB
searchengine.py	28-Mar-20 22:40	Python File	6 KB
structures_utils.py	27-Mar-20 0:48	Python File	3 KB

Figure 3.1 – E:\sri\ (the location of the project, before first execution)



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

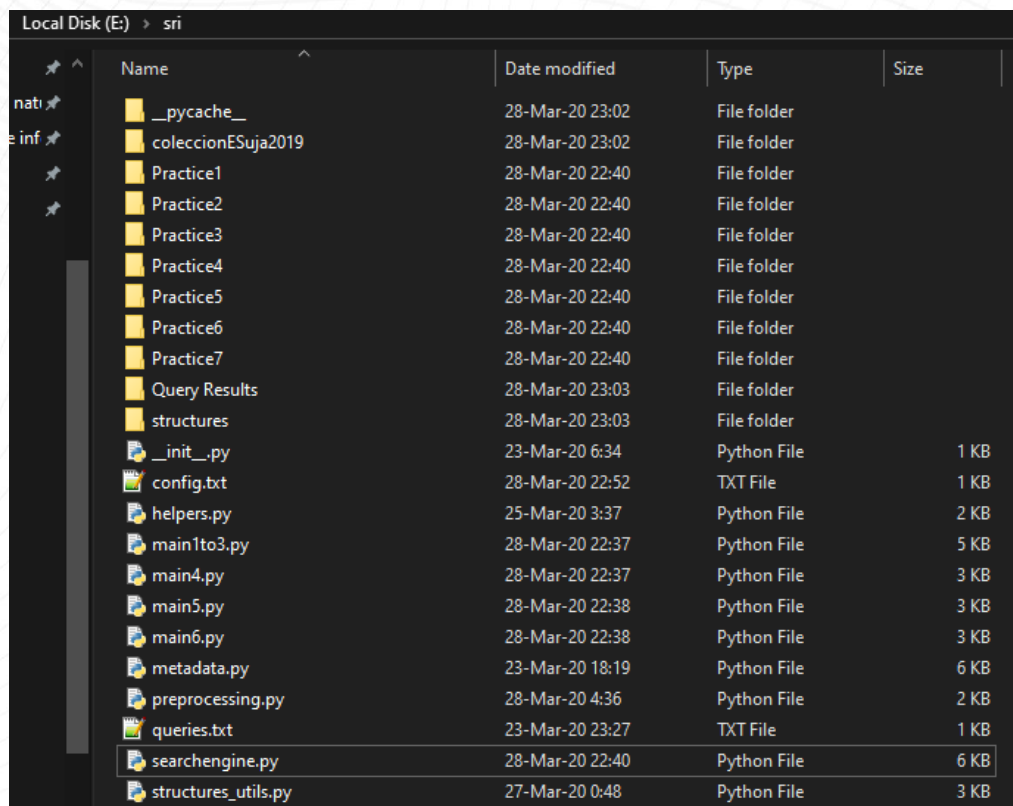
E:\sri>python searchengine.py E:\sri\config.txt E:\sri\queries.txt 10_
```

Figure 3.2 – How to execute the script

```
C:\Windows\System32\cmd.exe

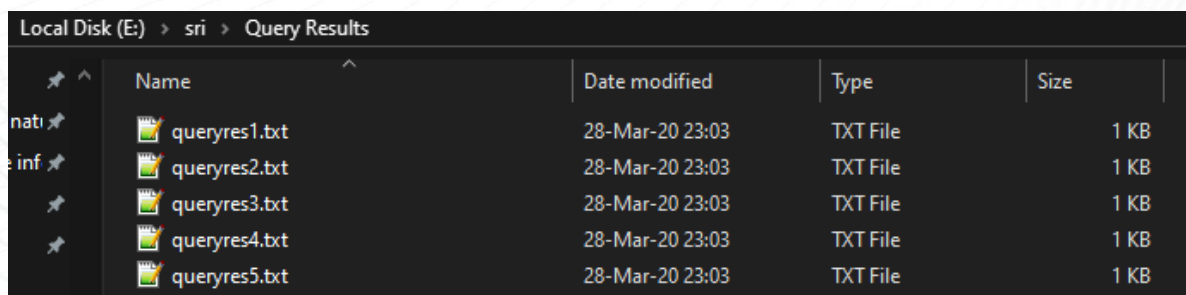
E:\sri>python searchengine.py E:\sri\config.txt E:\sri\queries.txt 10
Preprocessing time: 61.5802208s
Calculating time: 0.85673590000000039s
```

Figure 3.3 – Executing the script for the first time, index is created from the supplied collection



Name	Date modified	Type	Size
__pycache__	28-Mar-20 23:02	File folder	
coleccionESuja2019	28-Mar-20 23:02	File folder	
Practice1	28-Mar-20 22:40	File folder	
Practice2	28-Mar-20 22:40	File folder	
Practice3	28-Mar-20 22:40	File folder	
Practice4	28-Mar-20 22:40	File folder	
Practice5	28-Mar-20 22:40	File folder	
Practice6	28-Mar-20 22:40	File folder	
Practice7	28-Mar-20 22:40	File folder	
Query Results	28-Mar-20 23:03	File folder	
structures	28-Mar-20 23:03	File folder	
__init__.py	23-Mar-20 6:34	Python File	1 KB
config.txt	28-Mar-20 22:52	TXT File	1 KB
helpers.py	25-Mar-20 3:37	Python File	2 KB
main1to3.py	28-Mar-20 22:37	Python File	5 KB
main4.py	28-Mar-20 22:37	Python File	3 KB
main5.py	28-Mar-20 22:38	Python File	3 KB
main6.py	28-Mar-20 22:38	Python File	3 KB
metadata.py	23-Mar-20 18:19	Python File	6 KB
preprocessing.py	28-Mar-20 4:36	Python File	2 KB
queries.txt	23-Mar-20 23:27	TXT File	1 KB
searchengine.py	28-Mar-20 22:40	Python File	6 KB
structures_utils.py	27-Mar-20 0:48	Python File	3 KB

Figure 3.4 – Project folder after the execution (Query Results and structures folders are created)



Name	Date modified	Type	Size
queryres1.txt	28-Mar-20 23:03	TXT File	1 KB
queryres2.txt	28-Mar-20 23:03	TXT File	1 KB
queryres3.txt	28-Mar-20 23:03	TXT File	1 KB
queryres4.txt	28-Mar-20 23:03	TXT File	1 KB
queryres5.txt	28-Mar-20 23:03	TXT File	1 KB

Figure 3.5 – Query Results folder

queries1.txt - Notepad

File Edit Format View Help

"El olivar de Jaén."

1.	0.02684	es_46660.html
2.	0.02392	es_43864.html
3.	0.01990	es_43760.html
4.	0.01624	es_46294.html
5.	0.01565	es_45534.html
6.	0.01537	es_46301.html
7.	0.01514	es_45176.html
8.	0.01361	es_45608.html
9.	0.01176	es_46070.html
10.	0.01128	es_45543.html

Time taken: 0.00413s Number of similarities: 105

Figure 3.6 – Result file for the first query

Local Disk (E:) > sri > structures

Name	Date modified	Type	Size
fileref.sribk	28-Mar-20 23:03	SRI BK File	44 KB
index.sribk	28-Mar-20 23:03	SRI BK File	10,982 KB
wordref.sribk	28-Mar-20 23:03	SRI BK File	224 KB

Figure 3.7 – structures folder, files contained are created after the first script execution

Local Disk (E:) > sri > coleccionESuja2019	Local Disk (E:) > sri > coleccionESuja2019 > middle_output																								
<table> <thead> <tr><th>Name</th><th>Date modified</th></tr> </thead> <tbody> <tr><td>middle_output</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_26142.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43719.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43721.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43723.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43724.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43726.txt</td><td>04-Feb-19 17:30</td></tr> <tr><td>es_43728.txt</td><td>04-Feb-19 17:30</td></tr> </tbody> </table>	Name	Date modified	middle_output	28-Mar-20 23:02	es_26142.txt	04-Feb-19 17:30	es_43719.txt	04-Feb-19 17:30	es_43721.txt	04-Feb-19 17:30	es_43723.txt	04-Feb-19 17:30	es_43724.txt	04-Feb-19 17:30	es_43726.txt	04-Feb-19 17:30	es_43728.txt	04-Feb-19 17:30	<table> <thead> <tr><th>Name</th><th>Date modified</th></tr> </thead> <tbody> <tr><td>stage1</td><td>28-Mar-20 23:02</td></tr> <tr><td>stage2</td><td>28-Mar-20 23:02</td></tr> </tbody> </table>	Name	Date modified	stage1	28-Mar-20 23:02	stage2	28-Mar-20 23:02
Name	Date modified																								
middle_output	28-Mar-20 23:02																								
es_26142.txt	04-Feb-19 17:30																								
es_43719.txt	04-Feb-19 17:30																								
es_43721.txt	04-Feb-19 17:30																								
es_43723.txt	04-Feb-19 17:30																								
es_43724.txt	04-Feb-19 17:30																								
es_43726.txt	04-Feb-19 17:30																								
es_43728.txt	04-Feb-19 17:30																								
Name	Date modified																								
stage1	28-Mar-20 23:02																								
stage2	28-Mar-20 23:02																								

Figure 3.8 – middle_output folder inside the collection folder and its contents

Local Disk (E:) > sri > coleccionESuja2019 > middle_output > stage1	Local Disk (E:) > sri > coleccionESuja2019 > middle_output > stage2																																
<table> <thead> <tr><th>Name</th><th>Date modified</th></tr> </thead> <tbody> <tr><td>es_26142.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43719.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43721.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43723.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43724.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43726.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43728.txt</td><td>28-Mar-20 23:02</td></tr> </tbody> </table>	Name	Date modified	es_26142.txt	28-Mar-20 23:02	es_43719.txt	28-Mar-20 23:02	es_43721.txt	28-Mar-20 23:02	es_43723.txt	28-Mar-20 23:02	es_43724.txt	28-Mar-20 23:02	es_43726.txt	28-Mar-20 23:02	es_43728.txt	28-Mar-20 23:02	<table> <thead> <tr><th>Name</th><th>Date modified</th></tr> </thead> <tbody> <tr><td>es_26142.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43719.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43721.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43723.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43724.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43726.txt</td><td>28-Mar-20 23:02</td></tr> <tr><td>es_43728.txt</td><td>28-Mar-20 23:02</td></tr> </tbody> </table>	Name	Date modified	es_26142.txt	28-Mar-20 23:02	es_43719.txt	28-Mar-20 23:02	es_43721.txt	28-Mar-20 23:02	es_43723.txt	28-Mar-20 23:02	es_43724.txt	28-Mar-20 23:02	es_43726.txt	28-Mar-20 23:02	es_43728.txt	28-Mar-20 23:02
Name	Date modified																																
es_26142.txt	28-Mar-20 23:02																																
es_43719.txt	28-Mar-20 23:02																																
es_43721.txt	28-Mar-20 23:02																																
es_43723.txt	28-Mar-20 23:02																																
es_43724.txt	28-Mar-20 23:02																																
es_43726.txt	28-Mar-20 23:02																																
es_43728.txt	28-Mar-20 23:02																																
Name	Date modified																																
es_26142.txt	28-Mar-20 23:02																																
es_43719.txt	28-Mar-20 23:02																																
es_43721.txt	28-Mar-20 23:02																																
es_43723.txt	28-Mar-20 23:02																																
es_43724.txt	28-Mar-20 23:02																																
es_43726.txt	28-Mar-20 23:02																																
es_43728.txt	28-Mar-20 23:02																																

Figure 3.9 – contents of the stage1 and stage2 middle outputs

Improvements

The improvements that I have decided to implement for the second part of this project are:

- I/O Web interface
- Pseudo relevance feedback (PRF)

I have also removed the normalization from similarity calculations and fixed a bug concerning query words that are not in the index and a bug with queries that contain only stopwords.

Web interface

- API

Both the front-end and the API were developed using Flask, a microframework for Python. The API has 3 defined endpoints:

- /home or just / – which returns *main.html* with a *QueryForm* instance. This is the main entry point to the website.
- /search – which is used to send *QueryForm* data and the number of the desired page via a GET request, and then return the results in the JSON format. It takes the following arguments:
 - csrf – CSRF token
 - query – The query string
 - pagesize – Size of the pages that will be returned
 - fullpath – Whether to return a fullpath or just a filename for each result
 - prf – Whether to use PRF in the search
 - page (optional) – The number of the page request, if it is not specified, it returns the first page of the results
- /openfile – which is used to request the content of a specific .html file in the collection. It takes a sole GET argument, which is the *fileid* of the desired file.

Modules mentioned in this description:

- routes.py
- forms.py

QueryForm
+ query: StringField
+ pagesize: IntegerField
+ fullpath: BooleanField
+ prf: BooleanField
+ submit: SubmitField

- **Front-end**

There are two defined .html templates:

- *base.html* – which extends a flask-bootstrap template and defined the main areas in the page. The top part that contains the logo and the bottom part which contains the content.
- *main.html* – which defines the form area in the page and the element that will contain the table with the results. This is also where the script for AJAX requests is defined. One type of request for the initial form submission and one for the subsequent requests which are used to facilitate paging. I have used the *IntersectionObserver API* to implement an “infinite-scroll” like system. The last row of the results is observed and when it comes into view completely a callback is triggered. The results are drawn dynamically using the *DOM API*, and the drawing methods are defined in the static/resultsdisplay.js resource.

- **Configuration and execution**

The configuration settings are contained in the *config.py* module. The only important setting here is the `ENGINE_CONFIG`, which contains the path to the config file that is necessary for *SearchEngine* initialization. The contents of that config file are explained earlier in this document.

The modules that are necessary for the execution of this interface are:

- flask
- flask-wtf
- flask-bootstrap

I have written a couple of batch scripts to initialize a *virtualenv* and install the necessary packages and then run the server. The scripts are:

- *init.bat* – This only needs to be run once
- *start.bat* – Running this script hosts the server on the localhost

The scripts should be run in Command Prompt using the following command: `cmd /k <nameofthescript>`

- Web interface screenshots



Figure 4.0 – Welcome screen

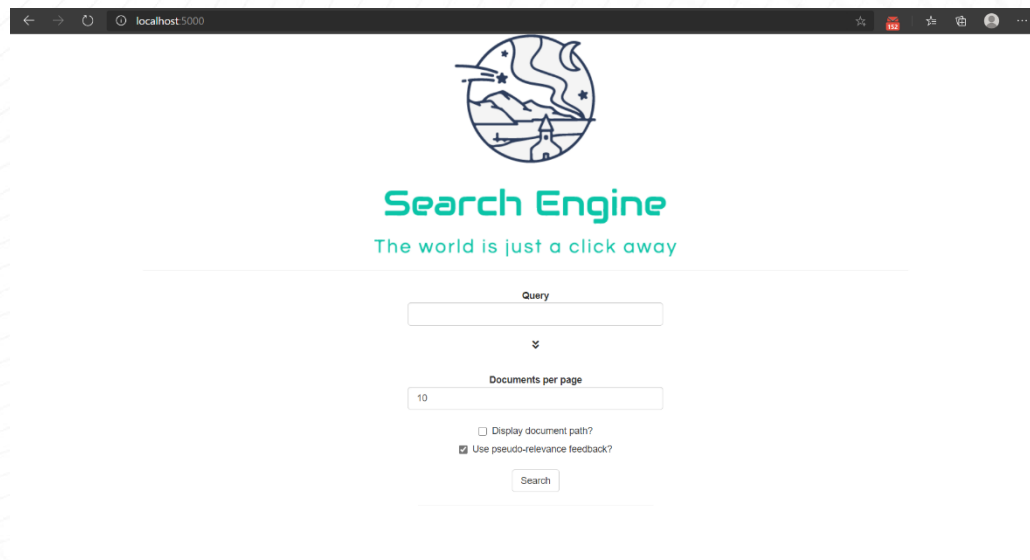


Figure 4.1 – Welcome screen with advanced search options open

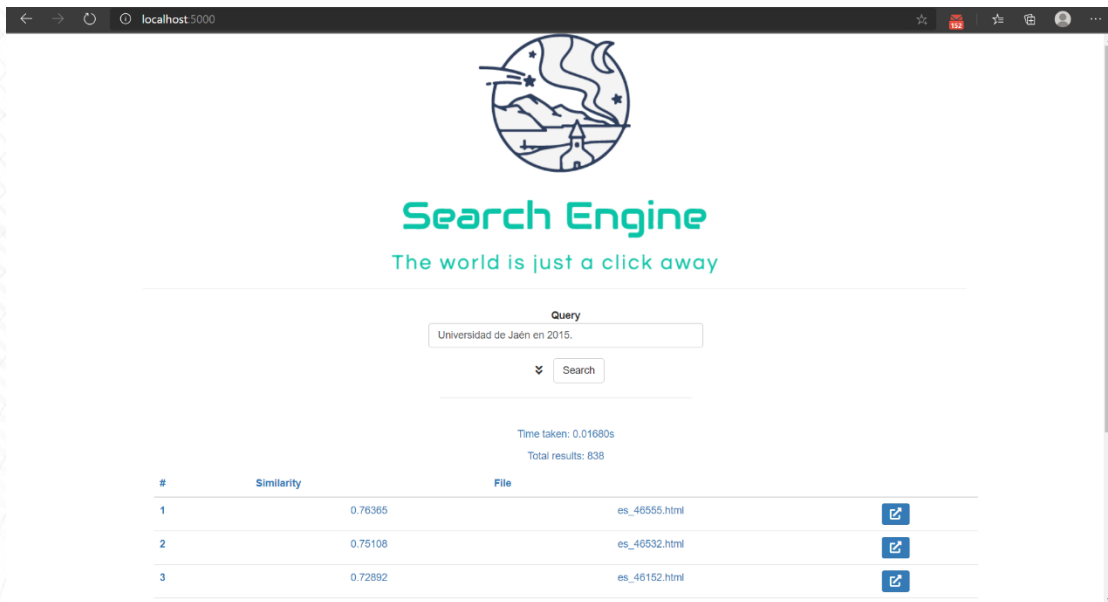


Figure 4.2 – Screen with search results. Each result has a button to open the file in a new tab.

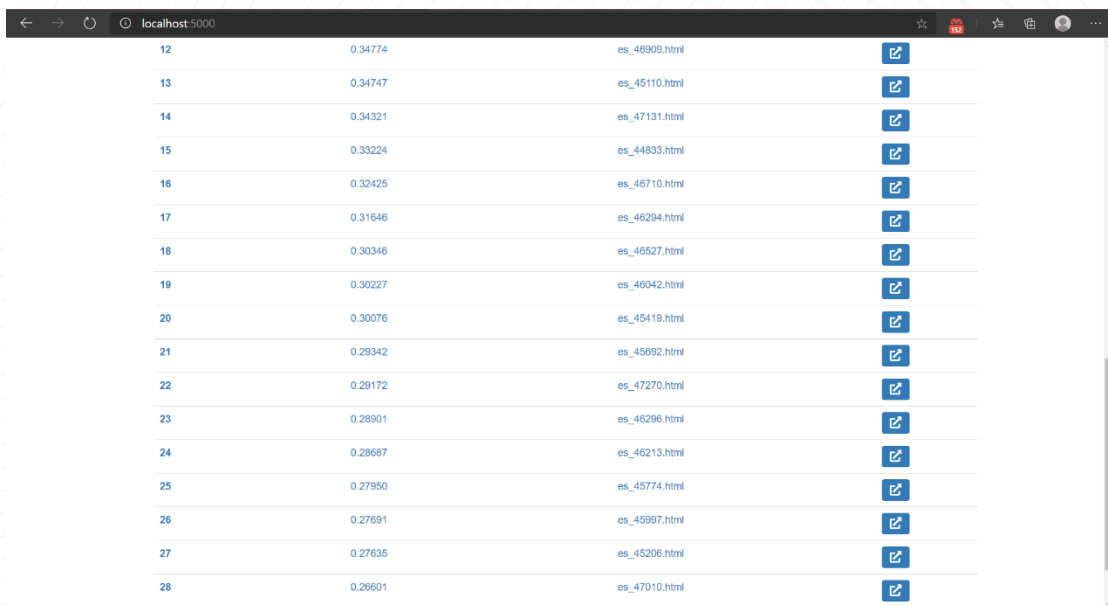


Figure 4.3 – Infinite scroll in action. Next results are fetched as soon as the user scrolls to the bottom of the page.

Pseudo relevance feedback

Implementation of PRF when searching the index required the addition of several methods and functionalities to some modules. Mainly the following:

- queryhandling.py
- similarity.py
- indexing.py

PRF is implemented as a rerun of the similarity search with added word tokens. We add N (default is 5) most common words from M (default is 5) files with the highest similarity value. To make this modification performant, I added a new method within the `IndexingData` class that generates a dictionary that holds pairs of a *fileid* and a sorted list of the N most common words for the file with that id. This method is called upon initialization of the `SimilarityGenerator` class, which saves the return structure and uses it in PRF expanded searches. I have also added functionalities to the `Query` and `QueryFactory` classes to allow the addition of tokens through PRF expansion, and the subsequent recalculation of weights.

```
def getSimilarities(self, query: queryhandling.Query, prf: bool = False, prffilecount: int = 5) -> Similarities:
    similarities = None

    starttime = timer()

    if prf and self.queryfactory:
        firstsimilarities = self.genSimilarities(query)
        firstsimilarities.sort()

        prfwordids = []
        for similarity in firstsimilarities.list[:prffilecount]:
            wordids = self.getPRFWordIDs(similarity.fileid)
            prfwordids.extend(wordids)

        newquery = self.queryfactory.addPRFWords(query, prfwordids)
        similarities = self.genSimilarities(newquery)
    else:
        similarities = self.genSimilarities(query)

    endtime = timer()
    similarities.setTimeTaken(endtime - starttime)

    return similarities
```

Figure 5.0 – Method in the `SimilarityGenerator` class that contains the main PRF logic.

Conclusion

I am pleased with the design of the project and its structure so far. Two functionalities that I would like to add are the addition of new files to the index and the expansion of the collection of stoppers and stemmers so that it can process multiple languages. I would also like to do more detailed tests on the performance of the index and try different optimizations, including multi-threading.

The goal of this project was to create a simple search engine. I believe that I have managed to accomplish this and create an adequate report along with it. My experience with Python before this was very limited and I can say that I have learned a lot during the development of this project.

Notes on improvements

The added web interface does its job well even though I would like to improve the design in the future. Overall, I think it functions well.

But, while PRF does what it is supposed to do, I am not completely convinced on whether it improves on the quality of the search. I think that a bigger collection would be needed to test that out.

I would have liked to have had the time to add the other proposed improvements (categories and snippets), so I might revisit this project in the future, as I believe it has been a very interesting experiment.