SEUPD@CLEF: Team XPLORE

Charles Milliaud[1], Manuel Antonutti[1], Aritz Plazaola Cortabarria[1] and Zain ul abedin Zain ul abedin[1]

[1]*University of Padua, Italy*

Abstract

This report outlines the efforts of Team XPLORE in response to the LongEval-Retrieval Task 1 Organizers [1] at CLEF 2024. The project aims to address the challenge of deteriorating performance in Information Retrieval systems over time intervals. By focusing on the development of a robust temporal IR system, we aim to enhance retrieval effectiveness despite temporal gaps between training and testing data. This report presents our methodologies, implementation details, and insights gained from our efforts to contribute to the LongEval Lab's.

Keywords

CLEF 2024, Information Retrieval, LongEval, Query expansion, Reranking

1. Introduction

In today's digital age, search engines have become indispensable tools, facilitating the retrieval of information essential to our daily lives. Yet, recent research has unveiled a critical challenge facing Information Retrieval (IR) systems: their performance tends to dicrease over time as the test data drifts further from the training data. This issue is especially pertinent in the dynamic landscape of computer science, where data undergoes constant updates, rendering once-relevant information obsolete. As such, the LongEval task 1 at Conference and Labs of the Evaluation Forum (CLEF) 2024 has gathered interest, aiming to assess the temporal persistence of IR systems.

Recognizing the pressing need for solutions to this challenge, our paper proposes an innovative approach. We seek to address this issue by developing a system capable of adapting to evolving data dynamics while maintaining high performance.

The data we will use is provided by the Qwant Qwant [2] search engine, encompassing user searches and web documents in both French and English. We posit that this robust dataset will empower our system to effectively adapt to shifts in user search behaviors and the evolving content landscape of web documents.

The paper is organized as follows:

- Section 2 describes our approach;
- Section 3 explains our experimental setup;
- Section 4 discusses our main findings; finally,
- Section 5 draws some conclusions and outlooks for future work.

"Search Engines", course at the master degree in "Computer Engineering", Department of Information Engineering, and at the master degree in "Data Science", Department of Mathematics "Tullio Levi-Civita", University of Padua, Italy. Academic Year 2023/2024

charlesedouard.milliaud@studenti.unipd.it (C. Milliaud); manuel.antonutti@studenti.unipd.it (M. Antonutti); aritz.plazaolacortabarria@studenti.unipd.it (A. P. Cortabarria); zainulabedin.zainulabedin@studenti.unipd.it (Z. u. a. Z. u. abedin)

2. Methodology

In this section, we describe the implementation details of our search engine and the approach we adopted to achieve the results listed in section 4. For the implementation of the XPLORE search engine we tried to follow a less is more approach, by focusing on the basics. Furthermore, by looking at the previous year LongEval competition results, we decided to use the French documents and disregard the English translations as they seem to be much less effective and tend to bring the score down.

2.1. Parser

The parser is the first main component of a search engine and its task is to clean up the documents before they are indexed. The documents were given to us by the CLEF organizers in two different formats: *JavaScript Object Notation (JSON)* and *Text REtrieval Conference (TREC)*. For this project, we preferred to adopt the TREC format, shown in Listing 1, because of its simplicity and understandability. Also, because we were provided with some examples of parsers already compatible with this format.

```
<DOC>
<DOCNO>  . . .  </DOCNO>
<DOCID>  . . .  </DOCID>
<TEXT>
   . . .
 </TEXT>
 </DOC>
```

Listing 1: TREC document format

Inside the parser package of the XPLORE search engine, we employed three main components that have two purposes. The first is to extract the document identifier and document body fields from the TREC files. The second is to polish the raw data by getting rid of html tags and meta-data. We inspected the collection manually and decided on deleting the html tags from the documents, since it could have had a bad impact on the search engine performance. It's important to note that although our approach was the safest, since it removed any chance of contamination from the web page elements, it also meant that we lost some meaningful data that could have used to improve our information retrieval system. For example, we could have used some of the html tags to understand its content and used different score weights for different parts of the file.

Here are the three main components of the parser package:

- **ParsedDocument**: this class represents the cleaned document after being parsed. It has two string attributes: the id and the body. It's used to store the retrieved documents ready to be indexed.

- **DocumentParser**: this component is an abstract class that provides the basic functionalities needed to parse the documents and to check if the parser has reached EOF for the current file.

- **LongEvalParser**: this is the main component of the parser package and its task is to extract the document fields (ID and BODY) from the TREC files, since the CLEF LongEval competition provides this format. It extends the DocumentParser class, so that it can be used by the indexer. It uses the basic String methods to extract the data from the files, while it depends on the Jsoup library to remove the html tags.

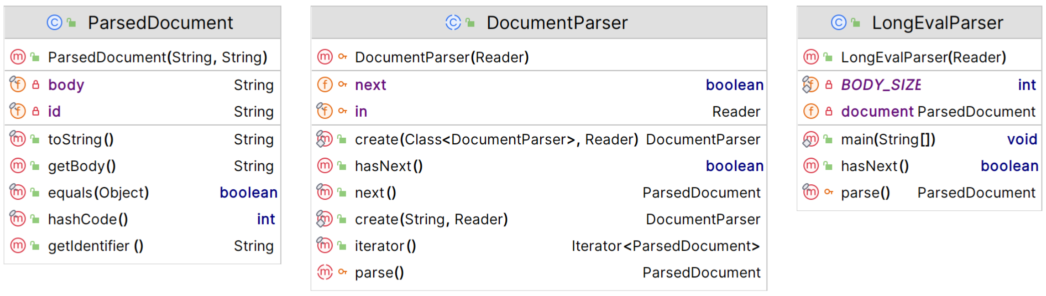

Figure 1: Class diagrams for the parser package

2.2. Analyzer

The analyzer is the second main component of a search engine and its task is to break down the documents into tokens and analyze them separately. This way each token can be standardized and prepared to be indexed. Many components can be chained to create a very complex analyzer stream, going from the basic tokenizers to more complex word N-grams filters and *Part of Speech (PoS)* taggers.
Inside the analyzer package of the XPLORE search engine, we employed three main components:

- **AnalyzerUtil**: this class has the helper methods used by the main analyzer to load files from the resource folder and other utilities. It can load stopword lists, lemmatizers, tokenizers and *Named Entity Recognition (NER)* taggers.

- **OpenNLPFilter**: this class is used to create a filter for the NER tagger of Apache OpenNLP. The TokenFilter you get with this class is not available in Lucene.

- **XploreAnalyzer**: this is the main analyzer class of the analyzer package. It reads the settings in the configuration file, using the ConfigReader object, and defines the token analysis stream accordingly. All the components defined in it are optional except for the tokenizer Lucene [3].

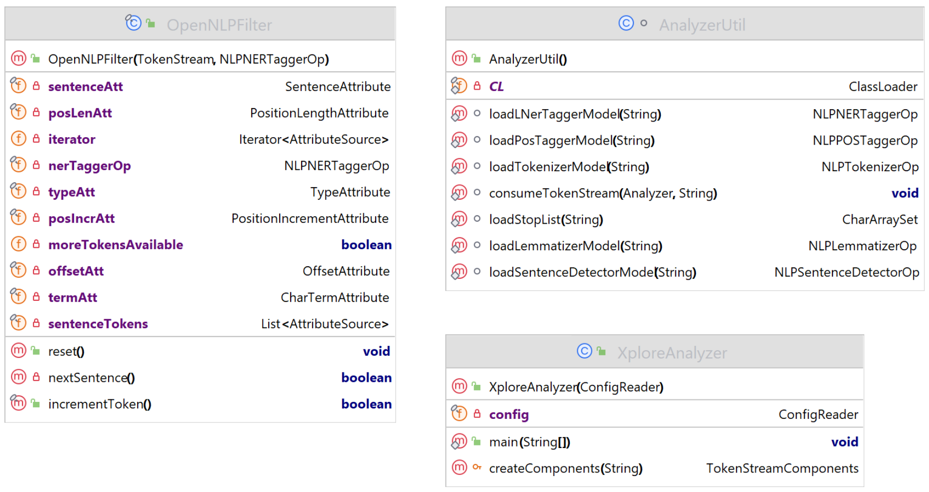
OpenNLPFilter
OpenNLPFilter(TokenStream, NLPNERTaggerOp)
sentenceAtt SentenceAttribute
posLenAtt PositionLengthAttribute
iterator Iterator<AttributeSource>
nerTaggerOp NLPNERTaggerOp
typeAtt TypeAttribute
posIncrAtt PositionIncrementAttribute
moreTokensAvailable boolean
offsetAtt OffsetAttribute
termAtt CharTermAttribute
sentenceTokens List<AttributeSource>
reset() void
nextSentence() boolean
incrementToken() boolean
AnalyzerUtil
AnalyzerUtil()
CL ClassLoader
loadLNerTaggerModel(String) NLPNERTaggerOp
loadPosTaggerModel(String) NLPPOSTaggerOp
loadTokenizerModel(String) NLPTokenizerOp
consumeTokenStream(Analyzer, String) void
loadStopList(String) CharArraySet
loadLemmatizerModel(String) NLPLemmatizerOp
loadSentenceDetectorModel(String) NLPSentenceDetectorOp
XploreAnalyzer
XploreAnalyzer(ConfigReader)
config ConfigReader
main(String[]) void
createComponents(String) TokenStreamComponents

Figure 2: Class diagrams for the analyzer package

The XploreAnalyzer component is in charge of processing the documents and is composed by:

- **Tokenization**: the tokenization is the process of splitting text into tokens of smaller length. Our search engine provides the choice between WhitespaceTokenizer Lucene [4], StandardTokenizer Lucene [5] and LetterTokenizer Lucene [6]. The first splits the text whenever it encounters a whitespace. The second one is a progression of the first, since it also divides the string into tokens whenever it sees a whitespace, but can split words at punctuation, by removing the punctuation. Although, if a dot is not followed by a whitespace, it interprets that as part of a token. The LetterTokenizer, instead, splits the text into tokens whenever it encounters a character that is not a letter. The idea behind this last tokenizer is that it defines tokens as maximal strings of adjacent letters.

- **Character folding**: ASCIIFolding and ICUFolding are used to apply search term folding to Unicode text. The first one, converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters into their ASCII equivalents, if one exists. Only those characters with reasonable ASCII alternatives are converted; whereas in the second one, applyies foldings from UTR#30 Character Foldings.

- **Elision removal**: the ElisionFilter allows us to remove the elision (the erasure of a vowel at the end of a word before the vowel beginning the following word) from a TokenStream.

- **Lowercase filtering**: the lowercase filter is used to turn all the uppercase letters into lowercase letters. We used the basic LowerCaseFilter provided by Lucene.

- **Token length filtering**: the token length filters are used to remove all the tokens that are outside a certain interval of length. By defining minimum and maximum length of the tokens, we can remove the outliers. Therefore, getting rid of probably uninformative terms. In our search engine, we used the basic LengthFilter provided by Lucene.

- **Stopword filtering**: the stopword removal is a process in which tokens that are considered useless or having very low information, are removed through the use of a stopword list. This list is composed of all the most prevalent stopwords in the language. The objective is to remove noise, save memory and improve the search engine performance from both an efficiency and an effectiveness standpoint.

- **Stemming**: the stemmer filters are mechanism to reduce the words into their root form. For example, the root form of *barking* is *bark*. In our search engine we provided the choice between FrenchLightStemFilter and FrenchStemmer.

- **Word N-grams**: The shingle size of word N-grams is configurable option for analyzer. It enhanced the functionality of search engine by introducing a configurable shingle size for word N-grams in the analyzer. This addition allows users to customize the size of word N-grams used during indexing and searching. By modifying the config.xml file, users can specify the desired shingle size, which is then read and utilized by the XploreAnalyzer class. This improvement enhances the flexibility and effectiveness of search engine, enabling users to fine-tune the analysis process according to the specific requirements.

- **Part of Speech tagging**: based on the definition and the context, it marks up a word in a text as a particular part of speech. This allows algorithms to understand the grammatical structure of a sentence and to disambiguate words that have multiple meanings.

2.3. Indexer

:Indexing is one of the most important steps in our search engine project. Here we create a searchable database that processes documents and makes them known as an index. Words and phrases are included in the index which contains vital information about the documents such as their frequency and where they appear in each document. Indexing provides us with an organized system enabling users to retrieve documents quickly by searching keywords or phrases. For this reason we have developed following components.

The indexer is the third main component of our search engine. It is an auxiliary access data structure and its task is to offer alternative methods to access data without impacting their physical layout. Indexing provides us with an organized system to retrieve documents quickly and efficiently.
Inside the indexer package of the XPLORE search engine, we employed the following two components:

- **DirectoryIndexer**: the DirectoryIndexer component in our search engine is the main class of the indexer package and is responsible for the indexing of documents. This component automates the process of traversing through the directory structure, identifying relevant documents based on their file extensions, and using the DocumentParser class described before to extract their contents. Then it instantiates a Document object with the extracted data and uses Lucene's IndexWriter to add it to the index. This process involves the use of the analyzer on the content to be indexed.

- **BodyField**: the BodyField class is used as a representation of the body content of a document and extends the Field class of Lucene. It can be configured so that the indexer stores this field for the purpose of reranking. It is important to note that storing the entire document body in the index causes the memory storage needed to rise by a lot.

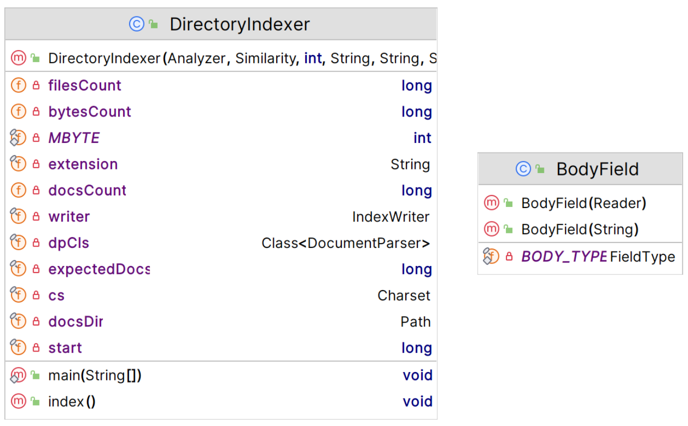

Figure 3: Class diagrams for the indexer package

Collection	Docs size (GB)	Stoplist	Stemmer	Documents (number)	Index size (GB)	Time (s)
French_lag6	8.10	Custom	FrenchLight	1593376	8.3	1907
French_lag8	5.57	Custom	FrenchLight	1081334	5.7	920

Table 1
Indexing time and storage needed for the LongEval test collections

2.4. Searcher

The searcher is the last component of a search engine and its task is to search the index to find the documents that are considered the most relevant to the query provided by the user. This searcher parses the queries and then applies an analyzer on them to better match the data inside the index. Note that it is not mandatory to use the same analyzer or similarity function for indexing and searching, but it is highly recommended. The searcher is a crucial part of any search engine and it's especially important that it is fast enough to be useful to the end user. Although it isn't one of the LongEval competition's main objectives, we decided that efficiency had not to be disregarded. Therefore, we tried to implement the searcher with this mindset.

Inside the searcher package of the XPLORE search engine, we employed three main components:

- **Searcher**: this is the main class of the package and it is the component used to search the index for the documents. It has a parser for the topics that reads the files in *Tab-Separated Values (TSV)* format. It uses either a BM25 or a LMDirichlet similarity function to initially score the document relevancy. Then it can employ more advanced techniques, like reranking, to attempt getting a better result.

- **Reranker**: this is the component that allows for the reranking of documents. This class implements a scoring algorithm that is based on the use of text embeddings, obtained from transformer models, and cosine similarity to determine the similarity between the representations.

- **SynonymMapper**: this is the component that allows the tokens of a query to be mapped to their synonyms. The synonyms are provided inside a synonym list file.

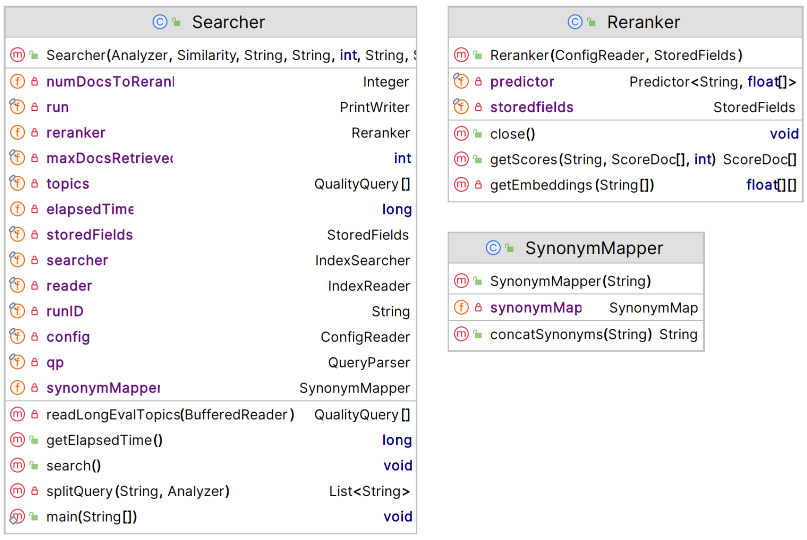
Searcher
Searcher(Analyzer, Similarity, String, String, int, String,
numDocsToReranl Integer
run PrintWriter
reranker Reranker
maxDocsRetrieve int
topics QualityQuery[]
elapsedTime long
storedFields StoredFields
searcher IndexSearcher
reader IndexReader
runID String
config ConfigReader
qp QueryParser
synonymMapper SynonymMapper
readLongEvalTopics(BufferedReader) QualityQuery[]
getElapsedTime() long
search() void
splitQuery(String, Analyzer) List<String>
main(String[]) void
Reranker
Reranker(ConfigReader, StoredFields)
predictor Predictor<String, float[]>
storedfields StoredFields
close() void
getScores(String, ScoreDoc[], int) ScoreDoc[]
getEmbeddings(String[]) float[][]
SynonymMapper
SynonymMapper(String)
synonymMap SynonymMap
concatSynonyms(String) String

Figure 4: Class diagrams for the searcher package

2.4.1. Main search

The basic search functionality has been implemented in the Searcher class of the searcher package. Its implementation involves the use of a Lucene BooleanQuery to make the matching of the query terms with those of the index very fast, especially when compared to alternatives like FuzzyQuery. The BooleanQuery has been configured with the clause parameter set to BooleanClause.Occur.SHOULD, so that it can retrieve documents even if they don't perfectly match all the terms in the query. Otherwise, the alternative BooleanClause.Occur.MUST would have made all the terms mandatory. The similarity function used to search for the documents in the index can be chosen from the settings in the *config.xml* file, inside the *resources* folder. The choice for the function is between the Okapi BM25 and the LMDirichlet, which we selected for their popularity in the field of information retrieval.

2.4.2. BM25

The Okapi BM25 (Best Matching 25) Lucene [7] is a probabilistic ranking function used to estimate the relevance of documents to a given search query. It is an extension of the TF-IDF (Term Frequency-Inverse Document Frequency) weighting scheme, which is the most popular at the moment.

BM25 includes k1 and b parameters that can be tuned to optimize the ranking function for a specific collection of documents and queries. These parameters control the relative importance of term frequency, document length, and term saturation in the ranking formula. We tried to maximize the nDCG measure while varying k1 and b values.

The default Lucene values were $k1 = 1.2$ and $b = 0.75$, and we tested for values of b between 0.6 and 0.9 and values of k1 between 1 and 2. The results can be seen in the table below.

		k1					
		1.0	1.2	1.4	1.6	1.8	2.0
b	0,60	0.3755	0.3761	0.3763	3765	0.3766	0.3768
	0,70	0.3773	0.3776	0.3801	0.3809	0.3812	0.3817
	0,80	0.3782	0.3797	0.3822	0.3825	0.3833	0.3840
	0,90	0.3803	0.3825	0.3830	0.3835	0.3840	0.3846

As can be seen, the best nDCG score is obtained with b=0.9 and k1=2. Therefore, we set the parameters with these values in the *config.xml* file. Also, it's important to note that these values have been measured early on in the development of the system, therefore the nDCG scores are lower than what they could be.

2.4.3. Query expansion

In information retrieval, query expansion is a technique used to improve the relevance of search results by augmenting or expanding the original user query with additional terms or concepts. The query expansion method that we used was mainly experimental. The first approach we decided to adopt was the one described in the article Desai [8] with some modifications. We wanted to extend each query with 10 to 15 related terms and see how the engine performed. To do so, we decided to use OpenAI's OpenAI [9] API and more precisely, GPT-3.5-turbo model to perform chat completion for us. In this scenario the input would be a prompt to extend the given query with 15 related terms. We would then write a script so that we can apply this technique on each line of the train.tsv file that was provided containing the queries for the train dataset.

After some tests using this technique we realised that doing perfect automatic query expansion without errors in the output file mainly generated by the LLM was impossible since we always had to manually adjust some errors in the .txt file (expansions given as a list with numbers or - to enumerate) so that it would match the desired .tsv query format.

Therefore, as a result, we decided to implement a simpler form of query expansion, even if less effective, using a simple synonym dictionary. In the Searcher class of the search engine, after performing an initial search with BooleanQuery Lucene [10], a rescore is performed (through the use of Lucene's QueryRescorer) that allows a new query to be used on the documents retrieved from the first step. The new query that is provided is a concatenation between the original query and a list of synonyms for all the terms contained in the query. The list of synonyms is obtained from the SynonymMapper component in the searcher package.

2.4.4. Reranking

Reranking is the process by which a score is reassigned to the documents obtained from the first phase of ranking. The reason for using this technique is to improve the result by using more complex algorithms compared to the first ranking, which instead uses a quick, but not too precise approach. In our search engine, we have decided to use transformer models that perform text embedding. In this way, documents and queries can be represented in the form of vectors. The similarity between two vectors is calculated with the cosine similarity function, chosen because of its popularity in the field. The combined score of the first ranking and the score of the cosine similarity are calculated using the equation (1), after being normalized in the range [0 to 1]:

$$CombinedScore = (1 - \alpha) \cdot Score_{Ranking} + \alpha \cdot Score_{Reranking} \tag{1}$$

The library used to load and employ the transformers is the *Deep Java Library (DJL)* DJL [11] that allows Java code to run models usually intended for Python. The models were downloaded from HuggingFace Face [12], a popular platform dedicated to *Artificial Intelligence (AI)*, and their names are: *dangvantuan/sentence-camembert-base*, *sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2*, *symanto/sn-xlm-roberta-base-snli-mnli-anli-xnli* and *sentence-transformers/all-MiniLM-L6-v2*. We could have used other strategies for reranking and more powerful models, like Llama or GPT-3, but we opted for this solution to improve the efficiency of the search engine.

2.5. Miscellaneous

In this subsection we describe some components of the XPLORE search engine that didn't belong to any specific group of functionalities, just for completeness:

- **ConfigReader**: this component is used to retrieve the general system settings to be used by the analyzer, indexer and searcher. The *config.xml* file inside the *resources* folder contains the xml entries of the configuration settings. Therefore, this class reads and parses the xml document and implements very simple get methods to access the data.

- **XploreSearchEngine**: this is the main class of the search engine, used to start both the indexing and searching processes. It reads the settings to be used in the run (like the indexPath, runPath, runID, etc) using the ConfigReader class.

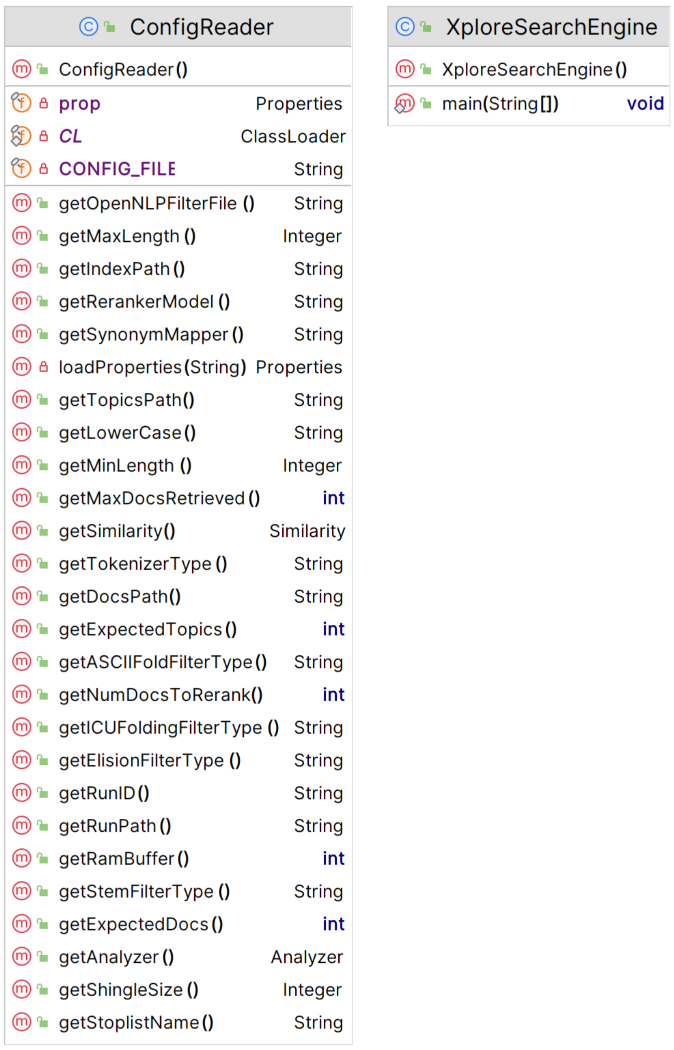

Figure 5: Class diagrams for the miscellaneous package

3. Experimental Setup

In this section we describe the following:

- Collection data used for training and testing
- Evaluation measures taken into consideration
- Git repository organization
- Hardware setup of our machines

3.1. Collection Data

To develop our search engine we used the data provided by the CLEF LongEval Lab, that is divided into one training collection and two test collections (*lag6* and *lag8*).

- **Training Data**: the collection consists of queries and documents provided by the Qwant search Engine https://www.qwant.com. The queries, which were issued by the users of Qwant, are based on the selected trending topics. The documents in the collection were selected with respect to these queries using the Qwant click model. Apart from the documents selected using this model, the collection also contains randomly selected documents from the Qwant index. All the data were collected over January 2023. In total, the collection contains 599 train queries, with corresponding 9,785 relevance assessments coming from the Qwant click model. The set of documents consist of 2,049,729 downloaded, cleaned and filtered Web Pages. Apart from their original French versions, the collection also contains translations of the webpages and queries into English. The collection serves as the official training collection for the 2024 LongEval Information Retrieval Lab https://clef-longeval.github.io/ organised at CLEF.

- **Testing Data**: The collection consists of queries and documents provided by the Qwant search Engine https://www.qwant.com. The queries, which were issued by the users of Qwant, are based on the selected trending topics. The documents in the collection were selected with respect to these queries using the Qwant click model. Apart from the documents selected using this model, the collection also contains randomly selected documents from the Qwant index. All the data was collected over June 2023 and August 2023. In total, the collection contains 1,925 test queries. The set of documents consist of 4,321,642 downloaded, cleaned and filtered Web Pages. Apart from their original French versions, the collection also contains translations of the webpages and queries into English. The collection serves as the official test collection for the 2024 LongEval Information Retrieval Lab https://clef-longeval.github.io/ organised at CLEF.

It is important to note that we decided to only use the French documents from the collections. This decision was made from the very start when we noticed the multiple translation errors found in the English data. But also, because by looking at the previous year's competition there was a clear correlation between the better performing systems and the ones using French data.

3.2. Evaluation measures

In order to determine the effectiveness of our system we relied on the following evaluation measures:

MAP: it is calculated by averaging the AP scores for multiple queries. It measures the ability of the system to retrieve relevant documents by taking into account the number of relevant documents in the ranked list.

$$MAP = \frac{1}{N}\sum_{i=1}^{N} AP_i \tag{2}$$

where N is the total number of topics. While the *Average Precision (AP)* score is calculated as:

$$AP = \frac{1}{RB}\sum_{k \in R} P(k) \tag{3}$$

where RB is the total number of relevant documents and R is the set of the rank positions of the relevant retrieved documents

nDCG: the Normalized Discounted Cumulative Gain is a measure that normalizes the *Discounted Cumulated Gain (DCG)* score of a query based on the maximum possible score it could reach, which is the *Ideal Discounted Cumulated Gain (IDCG)*.

$$nDCG@k = \frac{DCG@k}{IDCG@k} \tag{4}$$

3.3. Repository

The *git* repository of the project can be found at https://bitbucket.org/upd-dei-stud-prj/seupd2324-xplore/src/master/ and it is organized as follows:

```
/seupd2324-xplore/
├── code/
│   ├── src/main/
│   │   ├── java/it/unipd/dei/se/
│   │   │   ├── analyzer/
│   │   │   ├── indexer/
│   │   │   ├── parser/
│   │   │   ├── searcher/
│   │   │   ├── utils/
│   │   │   └── XploreSearchEngine.java <- Main class
│   │   ├── python/
│   │   └── resources/ <- config.xml, stoplists, synonym lists and others
│   └── pom.xml
├── homework-1
├── homework-2
└── runs <- run files submitted to CLEF
```

3.4. Hardware Setup

The search engine has been successfully run on the following setups:

- Charles Milliaud
  - OS: Windows 11
  - CPU: 13th Gen Intel(R) Core(TM) i7-1360P 2.20 GHz
  - GPU: None
  - RAM: 16 GB RAM
  - Storage: 1 TB SSD
- Manuel Antonutti
  - OS: Windows 10
  - CPU: 8th Gen Intel(R) Core(TM) i5-8265U 1.60 GHz
  - GPU: None
  - RAM: 8GB
  - Storage: 256 GB SSD
- Aritz Plazaola Cortabarria
  - OS: Windows 11
  - CPU: 12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz
  - GPU: None
  - RAM: 16GB
  - Storage: 1 TB SSD

4. Results and Discussion

In this section we provide some sample runs of the XPLORE search engine on the training data for this year's CLEF LongEval competition. We ran the search engine several times with different configurations, to see how it performed under different settings.

We have used keywords to refer to the configurations. The meanings of the keywords are as follows:

- French: means that the run used documents and queries from the French collection
- BM25Default: means that the run used Okapi BM25 as the similarity function with the default parameters
- BM25: means that the run used Okapi BM25 as the similarity function with the parameters specified in the section 2
- LMDirichlet: means that the run used the LMDirichlet function from Lucene
- Stop: means that the run used the custom french stopword list we added to the system
- FrenchLight: means that the run used the FrenchLightStemFilter from Lucene
- FrenchStemmer: means that the run used the FrenchStemmer from Lucene
- ASCII_Folding: means that the run used the ASCIIFoldingFilter from Lucene
- ICU_Folding: means that the run used the ICUFoldingFilter from Lucene
- Shingle3: means that the run used word N-grams with the size N set to 3
- Shingle5: means that the run used word N-grams with the size N set to 5
- Elision: means that the run the ElisionFilter from Lucene
- Reranker: means that the run used the Reranker component
- SynonymMapper: means that the run used query expansion with the SynonymMapper component

We have assigned a name to each of the run configurations to fit in the tables and thus improved comprehension of the data displayed. The names of the runs have the structure `seupd2324-xplore-X`. In addition, in the case of Table 3 we refer to the runs using `run_X` as a shorter identifier. In both cases `X` is the number of the run.

- run_1 = XPLORE_French—BM25—Stop—FrenchLight—ASCII_Folding—Shingle3
- run_2 = XPLORE_French—BM25—Stop—FrenchLight—ICU_Folding—Elision—Shingle5
- run_3 = XPLORE_French—BM25—Stop—FrenchStemmer—ASCII_Folding
- run_4 = XPLORE_French—BM25—Stop—FrenchStemmer—ICU_Folding—Elision—Shingle5
- run_5 = XPLORE_French—BM25—Stop—FrenchLight—ICU_Folding—Elision—Reranker
- run_6 = XPLORE_French—BM25—Stop—FrenchLight—ICU_Folding—Elision

In Table 2, we have ordered the runs from highest to lowest according to the MAP and nDCG values they obtained.

Table 2
The table with the MAP and nCDG values for each configuration in descending order

MAP and nDCG values of each run		
Configuration Name	MAP value	nDCG value
seupd2324-xplore-2	0.2763	0.4798
seupd2324-xplore-6	0.2754	0.4791
seupd2324-xplore-5	0.2723	0.4741
seupd2324-xplore-1	0.2696	0.4728
seupd2324-xplore-3	0.1933	0.3661
seupd2324-xplore-4	0.1908	0.3629

Table 3
Measures of each run with different configuration

Measures for French runs							
runID	all	run_1	run_2	run_3	run_4	run_5	run_6
num_q	all	584	585	583	581	597	597
num_ret	all	574119	576010	564765	562770	587728	587728
num_rel	all	4255	4261	4256	4235	4350	4350
num_rel_ret	all	3905	3939	3306	3321	4021	4021
map	all	0.2696	0.2763	0.1908	0.1933	0.2723	0.2754
gm_map	all	0.1329	0.1436	0.0269	0.0279	0.1459	0.1444
Rprec	all	0.2539	0.2581	0.1799	0.1802	0.2574	0.2572
bpref	all	0.5063	0.5066	0.4322	0.4326	0.4945	0.5049
recip_rank	all	0.4750	0.4873	0.3477	0.3535	0.4758	0.4845
iprec_at_recall_0.00	all	0.5147	0.5277	0.3769	0.3828	0.5214	0.5247
iprec_at_recall_0.10	all	0.5039	0.5159	0.3672	0.3732	0.5092	0.5131
iprec_at_recall_0.20	all	0.3826	0.3929	0.2763	0.2796	0.3893	0.3929
iprec_at_recall_0.30	all	0.3509	0.3613	0.2477	0.2527	0.3552	0.3596
iprec_at_recall_0.40	all	0.3042	0.3123	0.2117	0.2149	0.3056	0.3098
iprec_at_recall_0.50	all	0.2843	0.2908	0.1982	0.1994	0.2860	0.2897
iprec_at_recall_0.60	all	0.2317	0.2360	0.1593	0.1615	0.2351	0.2342
iprec_at_recall_0.70	all	0.1983	0.2024	0.1356	0.1374	0.2006	0.2010
iprec_at_recall_0.80	all	0.1812	0.1849	0.1244	0.1246	0.1831	0.1846
iprec_at_recall_0.90	all	0.1254	0.1278	0.0863	0.0878	0.1225	0.1270
iprec_at_recall_1.00	all	0.1205	0.1231	0.0830	0.0845	0.1178	0.1224
P_5	all	0.2634	0.2711	0.1877	0.1869	0.2690	0.2734
P_10	all	0.2284	0.2344	0.1612	0.1637	0.2308	0.2332
P_15	all	0.1998	0.2047	0.1401	0.1413	0.2008	0.2032
P_20	all	0.1737	0.1771	0.1224	0.1228	0.1771	0.1762
P_30	all	0.1377	0.1405	0.0975	0.0987	0.1406	0.1402
P_100	all	0.0559	0.0567	0.0413	0.0418	0.0566	0.0566
P_200	all	0.0304	0.0307	0.0232	0.0233	0.0308	0.0308
P_500	all	0.0129	0.0131	0.0104	0.0105	0.0131	0.0131
P_1000	all	0.0067	0.0067	0.0057	0.0057	0.0067	0.0067
recall_5	all	0.1871	0.1923	0.1314	0.1310	0.1933	0.1937
recall_10	all	0.3215	0.3302	0.2246	0.2279	0.3271	0.3291
recall_15	all	0.4182	0.4294	0.2905	0.2932	0.4241	0.4274
recall_20	all	0.4833	0.4929	0.3363	0.3370	0.4958	0.4915
recall_30	all	0.5722	0.5842	0.3999	0.4050	0.5863	0.5843
recall_100	all	0.7639	0.7750	0.5554	0.5624	0.7746	0.7746
recall_200	all	0.8263	0.8367	0.6234	0.6284	0.8385	0.8385
recall_500	all	0.8788	0.8896	0.7035	0.7082	0.8915	0.8915
recall_1000	all	0.9123	0.9189	0.7672	0.7725	0.9203	0.9203
ndcg	all	0.4724	0.4798	0.3629	0.3661	0.4741	0.4791

4.1. Considerations

Although it can be seen from the tables that the best nDCG score is reached by the `seupd2324-xplore-2` run configuration, we believe that `seupd2324-xplore-6` is better. The time needed by the `seupd2324-xplore-2` system to execute is much longer than time needed by the `seupd2324-xplore-6` system, almost double as long. This is not acceptable in a real case scenario where a user asks for a query and expects the results in a timely manner.

After running the system with different configurations, we have noticed that the `LightStemmer` is much more effective than the `FrenchStemmer`. Similarly, the `ICU_Folding` filter performs better than the `ASCII_Folding` filter. We have also assessed that the `Elision` filter helps to slightly improve the effectiveness of the system.

Another consideration after analysing the system is that the use of the `BM25` and `Stopword` list

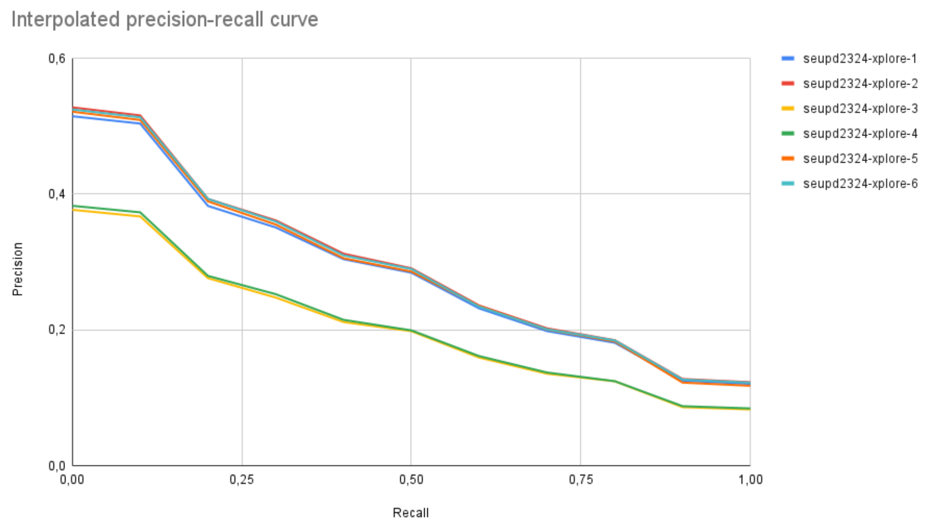

Figure 6: Interpolated Precision-Recall curves for each run configuration

removal is essential for the system, as it improves the scores of the system with very little computational overhead. The same cannot be said about `POS tagging` and `reranker`. Another improvement comes from the word N-grams, because by increasing their size the results improve, although, it also increases considerably the time needed to run the system.

Taking into account the above-mentioned considerations, we decided to submit the following systems:

- XPLORE_French-BM25-FrenchLight-Stop-SynonymMapper
- XPLORE_French-BM25-FrenchLight-Stop
- XPLORE_French-BM25Default-FrenchLight-Stop
- XPLORE_French-LMDirichlet-FrenchLight-Stop

As annotation, while choosing these systems we decided to use the LMDirichlet on one of the runs just to see what would happen, since, although we expect the system to perform poorly, there is a chance it could outdo BM25 on the testing data.

5. Conclusions and Future Work

Looking back at the results of our experiments, we can conclude that the proposed approach is simple, but effective.

We created the XPLORE search engine using the Lucene library as a backbone and implemented several well-established techniques to improve the search results. We first built the baseline, by assembling all the main components together and then added some of the more popular advanced strategies, like query expansion and reranking.

In the end, we managed to improve the MAP and nDCG scores overall. The best configuration from an effectiveness standpoint ended up being the one that used the BM25 as similarity function, the ICU folding filter as accents removal, the elision filter for the apostrophes, the stop-word removal filter, the French light stemmer provided by Lucene and the word N-grams with N set to 5. This system reached a MAP score of 0.2763 and a nDCG score of 0.4798. That said, it was also one of the slowest configurations to execute, being beaten only by the systems employing the reranker, that could take as long as 3 hours. Therefore, we believe that removing the word N-grams component makes for the best overall configuration, as the scores would only slightly worsen (less than 0.2% for nDCG) while the time to run would drop drastically (almost down to a half).

As part of future improvements, we aim to refine the components we've developed and extend the system's capability to process collections in languages beyond French. Another step in the right direction would come from figuring out how to constraint LLMs to only generate formatted output responses, maybe by fine-tuning a model or using specific tools like AzureOpenAI Azure [13] in JSON mode that can force a JSON file as output. This upgrade would mean improvements in both query expansion and reranking. The reranking based on text embedding models seemed to be working really well at first, but after improving the baseline model, by providing better accent filtering and stemming, the rerank scores went down. Therefore, exploring more sophisticated models beyond those previously tested may enhance the results.

References

[1] C. Organizers, Longeval clef 2024 lab, https://clef-longeval.github.io/, 2024. Accessed: 2024-05-20.

[2] Qwant, About qwant, https://about.qwant.com/en/, 2024. Accessed: 2024-05-20.

[3] A. Lucene, Lucene whitespacetokenizer, https://lucene.apache.org/core/7_4_0/analyzers-common/org/apache/lucene/analysis/core/WhitespaceTokenizer.html, 2024. Accessed: 2024-05-06.

[4] A. Lucene, Lucene whitespacetokenizer, https://lucene.apache.org/core/4_3_0/analyzers-common/org/apache/lucene/analysis/core/WhitespaceTokenizer.html, 2024. Accessed: 2024-05-06.

[5] A. Lucene, Lucene standardtokenizer, https://lucene.apache.org/core/6_6_0/core/org/apache/lucene/analysis/standard/StandardTokenizer.html, 2024. Accessed: 2024-05-06.

[6] A. Lucene, Lucene lettertokenizer, https://lucene.apache.org/core/7_3_1/analyzers-common/org/apache/lucene/analysis/core/LetterTokenizer.html, 2024. Accessed: 2024-05-06.

[7] A. Lucene, Lucene bm25similarity, https://lucene.apache.org/core/7_0_1/core/org/apache/lucene/search/similarities/BM25Similarity.html, 2024. Accessed: 2024-05-06.

[8] A. A. Desai, Improving rag with query expansion & reranking models, medium.com (2024).

[9] OpenAI, Openai text completion api documentation, https://platform.openai.com/docs/guides/completion/introduction, n.d. Accessed: 2024-05-06.

[10] A. Lucene, Lucene booleanquery, https://lucene.apache.org/core/8_1_1/core/org/apache/lucene/search/BooleanQuery.html, 2024. Accessed: 2024-05-06.

[11] DJL, Djl - deep java library, 2024. URL: https://djl.ai/, accessed: May 6, 2024.

[12] H. Face, Hugging face – the ai community building the future, 2024. URL: https://huggingface.co/, accessed: May 6, 2024.

[13] M. Azure, Azure openai service – advanced language models, 2024. URL: https://azure.microsoft.com/en-us/products/ai-services/openai-service.