SEUPD@CLEF: Team Squid on LongEval-Retrieval

Notebook for the LongEval Lab on Longitudinal Evaluation of Model Performance at CLEF 2023

Vittorio Cardillo[1], Alberto Dorizza[1], Mattia Maglie[1], Dario Mameli[1], Gianluca Rossi[1], Michele Russo[1] and Nicola Ferro[1]

[1]*University of Padua, Italy*

Abstract

This is the report based on the work done for LongEval Task 1: Retrieval at CLEF 2023, by team Squid (whose participants are from the University of Padua).

Information retrieval (IR) systems have played an increasingly important role in our society and people's daily lives. Although they have become more and more powerful during the last decades, their temporal persistence is still causing drops in performance, thus failing to achieve good temporal generalisability. To investigate and improve the resolution of this issue, in this paper, we present and discuss the various solutions submitted to the first CLEF 2023 shared task (LongEval-Retrieval), which precisely requires the development of temporal information retrieval systems.

Keywords

Search Engine, Information Retrieval, Query Expansion, Query Boost, Word2Vec

1. Introduction

In this report, we will describe in detail the approach our group followed in the development of a solution to the Long-Eval Retrieval task by *Conference and Labs of the Evaluation Forum (CLEF)* [1], examining the results we obtained. Our group is formed by students attending the Search Engines course a.y. 2022/23 at the Computer Engineering Master degree at the University of Padua.

The aim of the Long-Eval Retrieval task is to develop information retrieval systems that can provide a resolution of the temporal persistence problem in *Information Retrieval (IR)*. In fact, despite the advancements of IR systems, their resilience to time remains a challenge, leading to drops in performance and hindering their ability to achieve good long-lasting generalizability.

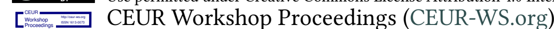
Our paper is organized as follows:

CLEF 2023: Conference and Labs of the Evaluation Forum, September 18-21, 2023, Thessaloniki, Greece

vittorio.cardillo@studenti.unipd.it (V. Cardillo); alberto.dorizza@studenti.unipd.it (A. Dorizza); mattia.maglie@studenti.unipd.it (M. Maglie); dario.mameli@studenti.unipd.it (D. Mameli); gianluca.rossi.4@studenti.unipd.it (G. Rossi); michele.russo.2@studenti.unipd.it (M. Russo); ferro@dei.unipd.it (N. Ferro)

http://www.dei.unipd.it/~ferro/ (N. Ferro)

0000-0001-9219-6239 (N. Ferro)



© 2023 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3497/paper-190.pdf

- Section 2: here we briefly explain the background knowledge and the starting code on which we built the systems.
- Section 3: here we describe the workflow of our IR system by examining its' modules and their functioning, delving deeply into the code that is the core of our systems.
- Section 4: here we explain our experimental setup, describing the data, the evaluation measures, the structure of the repository, and the hardware we used to run the system.
- Section 5: here we discuss our main findings and compare the different systems in terms of their performance with respect to the evaluation measures taken into consideration.
- Section 6: here we perform the statistical analysis of the systems using different tools on the basis of their performances on the test sets.
- Section 7: here we draw our final conclusions and propose some improvements to the systems, to better meet the goals of the proposed task.

2. Related Work

The systems we are going to describe in Section 3 are based on different elements that were presented during the Search Engines course, held in this academic year (2022/23). We examined different papers and implementations for further insight into the topics of the course and more advanced techniques.

2.1. Starting code

We started our work by examining the code that was showcased during the Search Engines course (2022/23). The following classes were provided: *ParsedDocument*, *DocumentParser*, *TipsterParser*, *DirectoryIndexer*, *BodyField*, *TopicsReader*, *Searcher*. These classes are primarily based on the Java library of *Apache Lucene*.

The details on the adaptation of these classes are provided in Section 3.

2.2. External works

In our systems, we have used different tools which belong to the scientific literature or public domain.

We employed Okapi *Best Match 25 (BM25)* and *Term Frequency–Inverse Document Frequency (TF-IDF)* as similarity functions for the *Indexing* and *Searching* phases. For the *query expansion* part we experimented on the generation of synonyms using *WordNet* thesaurus [2], as well as by using locally trained neural models like *Word2vec* [3], and via Java API of publicly available generative models like GPT-3.5. We also experimented with *flair* [4] and `fr_core_news_sm` for the *Named Entity Recognition (NER)* task.

3. Methodology

In this section we describe the methodology we adopted, taking a closer look at the overlying architecture of our systems, and their workflows. The description we provide is general, and it examines all the components that make up our systems, which will all be reported in Section 5.

Starting from the powerful Java library *Apache Lucene*, we experimented with different applications of *Natural Language Processing (NLP)*, Machine Learning, and Deep Learning, to enhance the systems' performances.
By examining a wide variety of libraries implemented both in Java and Python, we have constructed multifaceted systems consisting of both independent and interdependent modules, ensuring great versatility in usage.

All the systems' implementations follow a standard pipeline which can be summarized in the following way:

- **Parser**: process of parsing the *HyperText Markup Language (HTML)* documents
- **Analyzer**: the process of analyzing the parsed texts of the documents
- **Indexer**: the process of indexing the documents' content, using the analyzer
- **Searcher**: the process of searching the relevant documents at query time
- **Re-ranking**: the process of ordering the documents depending on the relevance

While Parser and Indexer follow the same pattern across all systems, Analyzer, and Searcher have been tested with different implementations.

3.1. Parser

In this subsection, we describe the pipeline for parsing the documents of the corpora provided by CLEF.
By manually inspecting the collection, we found that many documents could be well represented by their first words. Therefore, we added a field called *title* to the *ParsedDocument*, where we intended to put those first words, for each document. The other field is *body*, where we intended to put the whole parsed content of the document, and *ID*, where the document ID is stored.

The *SquidParser* is a modified version of the *TipsterParser*. The core functionality is therefore kept equal, but some tweaks were applied to take into account the expansion of the *ParsedDocument* class. *SquidParser* extends the *DocumentParser* which is an abstract class whose extensions implement the iterator interface over *ParsedDocument* type objects. This allows *SquidParser* to iteratively parse all documents in the collection and create a *ParsedDocument* object for each document.

In particular, each document is stripped of the HTML tags and the ID is stored as it is in the respective field. The remaining words are saved in the *body* field. The first 60 characters of the cleaned document are also saved in the *title* field, however, to avoid saving cut words at the 60th character, we allow some tolerance by saving the whole final word.

This *ParsedDocuments* will be useful in the indexing phase, as will be explained in Section 3.3.

3.2. Analyzer

In this subsection, we discuss the analysis methods and instruments we used in order to find the best combination to process documents and queries. The analyzer is one of the most important parts of an Information Retrieval system, it concerns with pre-processing text to create a most fundamental token stream, that will be indexed.

Analyzer has three main parts:

- Tokenizer that divides a sentence into smaller parts called tokens
- Stop Words removal that removes common words according to a stoplist
- Stemming that removes prefixes and suffixes by picking a common stem for a token

Since the Analyzer is able to delete tokens (using a stop list) and modify tokens, little modifications can have a big impact in terms of performance.

3.2.1. Document inspection

Inspecting the collection is the first step to creating a good Analyzer.

By looking into documents, we have seen that they do not have a "native" structure given by tags, which means that they do not have any *title* tag nor any other.

We provided to structure documents into the parser, by adding the first 60 characters of each document into the Title field. This is a good trade-off, because of the absence of a title tag we are not able to understand clearly when the title ends, but by an accurate inspection and looking at many runs, we have understood that 60 characters are enough to take only the title and have good performance.

Moreover, we thought it would be appropriate to insert a new field for the *Uniform Resource Locator (URL)* since all documents use them, and since we will then boost at query time all the documents that have a URL field whose tokens match with those of the query.

For this purpose, we used the "url.txt" file given by CLEF.

Finally, we changed the representation of characters when we open file documents to UTF-8, in order to correctly read French characters.

3.2.2. Stop lists

We tested the performance of the system with many stoplists in order to understand better how much stopword removal can change the performance. First of all, we tested all the stop lists mentioned during the lectures:

- atire.txt[1] - 988 words
- terrier.txt[2] - 733 words
- smart.txt [3] - 571 words
- zettair.txt[4] - 469 words
- indri.txt [5] - 418 words
- glasgow.txt[6] - 319 words
- okapi.txt[7] - 222 words
- snowball.txt[8] - 174 words
- lucene.txt[9] - 33 words

After inspecting via *Luke* (Lucene's open-source tool used for inspecting and debugging Lucene indexes) the index generated without using any document pre-processing (e.g stoplist, stemmer, tokenizer), we created and tested two stoplists consisting of:

- the first 100 words with the highest frequency in the index
- the first 200 words with the highest frequency in the index

After seeing that performance with the English language did not increase, and since the given dataset also contains the original documents in French, we tried the following French stoplists:

- stoplist_fr_691.txt[10] - 691 words
- stoplist_fr_496.txt[11] - 496 words

[1]https://github.com/andrewtrotman/ATIRE/blob/master/source/stop_word.c
[2]http://terrier.org/docs/current/javadoc/org/terrier/terms/Stopwords.html
[3]http://ftp.sunet.se/mirror/archive/ftp.sunet.se/pub/databases/full-text/smart/english.stop
[4]http://www.seg.rmit.edu.au/zettair/download.html
[5]http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words
[6]http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words
[7]http://www.staff.city.ac.uk/~andym/OKAPI-PACK/appendix-d.html
[8]https://github.com/snowballstem/snowball-website/blob/master/algorithms/english/stop.txt
[9]https://github.com/apache/lucene/blob/main/lucene/analysis/common/src/java/org/apache/lucene/analysis/en/EnglishAnalyzer.java
[10]https://github.com/stopwords-iso/stopwords-fr/blob/master/stopwords-fr.txt
[11]https://countwordsfree.com/stopwords/french

- stoplist_fr_463.txt[12] - 463 words
- stoplist_fr_nltk.txt[13] - 247 words

After testing the standard French stoplists, we generated two more stoplists using index inspection with Luke, as we did for the English language:

- the first 100 French words with the highest frequency in the index
- the first 200 French words with the highest frequency in the index

3.2.3. FrenchAnalyzer

The analyzer that we mostly used is the `FrenchAnalyzer`[14] in the org.apache.lucene.analysis.fr package of Lucene's standard library[15].

This class is composed by:

- *StandardTokenizer*
- *StandardFilter*: normalizes the tokens from *StandardTokenizer*.
- *ElisionFilter*: removes elisions.
- *LowerCaseFilter*: sets tokens to lowercase.
- *StopFilter*: with *Snowball*'s stopwords[16].
- *KeywordMarkerFilter*: marks terms as keywords via the *KeywordAttribute*.
- *SnowballFilter*: a filter that stems words using a *Snowball-generated* stemmer.

actually it is the one which allows us to have the best performances

3.2.4. Custom Analyzer

The Custom Analyzer developed is based on the French Analyzer of the Lucene library.

Implementation It uses the *LetterTokenizer* instead of the *ClassicTokenizer* of Lucene. Our developed Custom Analyzer uses both *FrenchLightStemFilter* and *FrenchStemmer* based on the *SnowballFilter*, where the latter attempts to remove common suffixes from words but may also introduce some inaccuracies or over-stemming in certain cases.
Our thought was that by using both, FrenchLightStemFilter and FrenchStemmer, we could achieve higher performance than the FrenchAnalyzer.

[12] https://sites.google.com/site/kevinbouge/stopwords-lists
[13] https://github.com/Alir3z4/stop-words/blob/bd8cc1434faeb3449735ed570a4a392ab5d35291/french.txt
[14] https://lucene.apache.org/core/5_0_0/analyzers-common/org/apache/lucene/analysis/fr/FrenchAnalyzer.html
[15] https://lucene.apache.org/core/9_5_0/index.html
[16] https://snowballstem.org/

French Analyzer As reported in Section 3.2.3, the French Analyzer only uses the *French-LightStemFilter* to stem words. It aims to preserve more of the original word forms while still reducing words to a common base form.

3.3. Indexer

In this subsection, we describe how the indexing is carried out.

The index we build via Lucene's *IndexWriter*[17] is a traditional inverted index, which is used in most information retrieval systems. In particular, each entry is described by a field, and there are as many fields as in the *ParsedDocuments*, from which we want to extract the content.

The *IndexWriter* is also built with some specified configurations, which include an analyzer and a similarity measure. We tested different analyzers and similarities, whose combinations will be addressed in Section 5.

We build a Lucene document with fields *ID*, *BodyField*, *TitleField*, for each *ParsedDocument*, where we put the content as it is from the respective fields in the *ParsedDocument*.

For each Lucene document, we have decided to assign the respective URL as *UrlField*. To use them we have implemented the *ParsedUrls* class. This class has the task of parsing the URLs and storing them in a hashmap, separating each word that composes them by removing the special characters.

These Lucene documents are written by the *IndexWriter* in the actual inverted index in the following way:

- *ID* content as it is in the respective field.
- *BodyField*, *TitleField*, *UrlField* contents after processing by the specified analyzer, as tokens, in the respective fields.

For the fields *BodyField* and *TitleField*, *IndexWriter* is also tasked to save not only the references to the documents which contain each token of said field, but also the relative frequencies of the terms within each referenced document and the positions of said terms, to allow proximity-based search and phrase matching, which is then used at query time in the *SearcherW2V* class.

For the field *UrlField*, it only saves the reference to the document which contains each token and the relative term frequency.

3.4. Searcher

In this subsection, we discuss the different implementations of the Searcher, following different approaches. These implementations have been used to produce the runs independently of each other, meaning, each one is part of one system only, as described in Section 4.4.

For reading the topics, the custom *MyTopicsReader* has been developed on the basis of the original *TopicsReader*, to account for the lack of *narrative* and *description* fields in the topics given by CLEF.

[17] https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/index/IndexWriter.html

3.4.1. BasicSearcher

Starting from the simplest Searcher, *BasicSearcher* is composed of the following steps:

- build one boolean query from the original query string via Lucene's *QueryParser*[18], for the *body*, *title* and *URL* fields of the index, using *SHOULD* clause.
- boost each query with a specific value depending on the field of the index to which it will be matched.
- concatenate the boosted queries into a single *Query* using *SHOULD* clause.
- search the documents using Lucene's *IndexSearcher*[19], considering this final *Query*.

The process is obviously iterated for each query given in the dataset.

3.4.2. SearcherBoost

The *SearcherBoost* class has the goal to boost the terms of the given queries, using the *Inverse Document Frequency (IDF)* of each term. The Search method that performs the actual search of the given queries, uses four different components: *printVocabularyStatistics*, *rareWords*, *boostQuery*, and *boost*.

printVocabularyStatistics This method is responsible for scanning the index for computing the raw frequency and document frequency of each term. Each term is saved in a hashmap as an entry of *<String,long[]>*, where String is the term and the array *long[]* contains the raw frequency and document frequency. When the computation is done, the result is saved in a text file. Moreover, to perform the IDF, we save the term and a tuple containing the raw frequency and document frequency into a new hashmap; here we save only the terms that have a document frequency below a set threshold: this is necessary for collecting only the discriminating terms of the document collection.

rareWords rareWords method saves the term and its document frequency in the hashmap.

boostQuery boostQuery method is invoked for each input query. Every query is transformed into an array of String, then each String is converted into a Query and boosted based on its IDF value. In the end, the query terms are gathered together with the Boolean query using the Boolean clause *SHOULD*.

boost Here we compute the IDF boost using the IDF formula $\log_2 \frac{N}{n_i}$, where N represents the number of documents in the collection and n_i represents the document frequency of that term.

[18]https://lucene.apache.org/core/8_0_0/queryparser/org/apache/lucene/queryparser/classic/QueryParser.html
[19]https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/search/IndexSearcher.html

3.4.3. SearcherW2V

Rationale The task of performing effective query expansion is not trivial. After an unsuccessful first draft using ***WordNet*** [2] for both the English-translated corpus and the French one, we were looking for alternative models. *WordNet* is a thesaurus, and as such, it has no knowledge about the collection. Therefore it generates synonyms that might be similar to the given word, but unrelated to the distribution of words in the collection or not present at all in it.

Given the better performances obtained with the French collection overall, and given the lack of pre-trained models for **synonym expansion** for the French language, we decided to look for a machine learning model that could be trained directly on the French collection. However, we wanted it to generate synonyms that are also "context-aware" in a sense, that is, that might be found close to the given word, and surely within the collection, with the ultimate goal of improving recall.

To this end, we came across ***Word2vec*** [3], which we are going to use for synonyms expansion, as will be thoroughly explained in the *Query Expansion* paragraph (3.4.3).

Training *Word2vec* has been implemented both in Java, under the *Deeplearning4j* library, and in Python as a *Gensim Word2vec model*. For training purposes, we used the *Gensim Word2vec model*, and the best parameters we have found are:

- `sentences`: set to a custom-defined sentence iterator which is able to iterate through all the files in the collection path, preprocessing each sentence by stripping tags, punctuation, and unnecessary symbols, and returning a list of tokens by analyzing the preprocessed sentence with the french tokenizer and french stoplist given by the *Natural Language Toolkit (NLTK)* python library.

- `vector_size=300`: dimensionality of the word vectors.

- `window=8`: maximum distance between the current and predicted word within a sentence. Bigger windows allow us to better represent sentences, however, due to stopwords removal, we thought we could afford to keep the size relatively low.

- `min_count=10`: ignores all words with total frequency lower than this. We found that roughly 90 percent of the words in the corpus have less than frequency 10 in the whole collection. This big reduction allows for better characterization of the remaining words, which still sum up to almost a million.

- `sg=1`: training algorithm: 1 for *skip-gram*; otherwise *Continuous Bag of Words (CBOW).*

- `negative=20`: if > 0, *negative sampling* [5] will be used, the int for negative specifies how many “noise words” should be drawn (usually between 5-20). If set to 0, no negative sampling is used. It is basically used to change the weights of the network in an SGD-like fashion, depending on this number of observations.

- `epochs=5`: number of epochs of training over the corpus.

A significant contribution to the choice of the parameters was given by [6].

Named Entity Recognition ***Named entities*** are words that represent real-world "objects", such as a person, location, organization, product, etc., that can be denoted with proper nouns.

Since Named Entities refer to words of common interest we did not want to expand them by finding synonyms, so we provided an algorithm of **NER** in *fr_core_news_sm.ipynb* , in order to detect *Named Entities* in queries and insert them into a txt file.

In practice, we loaded a pre-trained model called ***fr_core_news_sm***[20]. This model is trained using the *Le Monde* dataset, which is a French text corpus of approximately 45 million tokens. The corpus contains news articles from the French newspaper *Le Monde*, with coverage ranging from 1987 to 2017. The corpus has been annotated with linguistic information such as *parts of speech* and *named entities* using the *Universal Dependencies* [7] annotation framework.

In practice, feeding the model with a query text will provide as output the name of the entities, if present, which we can then use in the following query expansion.

Another French *NER* we had looked into is ***Flair*** [8, 4].

Flair is a *NER* model trained using a document from *Wikipedia 2019*, and it has many language implementations. The French *Flair* implementation fine-tunes the original one, which is a multi-language model, by using *Le Monde* and *European Parliament Proceedings Parallel Corpus 1996-2011* [9].

Both the implementations are valid, but after an accurate inspection of the results we preferred *fr_core_news_sm*, because *Flair* is computationally too expensive for our hardware.

Moreover, due to the fact that it starts as a multi-language model, it struggles to find some named entity, especially when all the words are lowercase.

Therefore, in order to exploit all its potential, it required better training, which we could not afford due to hardware limitations.

Query expansion After training the *Word2vec* model we wanted to use it for the *query expansion* task of generating synonyms for specific terms of the query. We used the *Deeplearning4j* library to load the *Word2vec* model in *SearcherW2V*. From this point, we modified the original *Tipster Searcher* in the way queries are generated, by having for each topic a single final *Query* made by concatenating in OR the following queries:

- **original query**:
  1. preparse the *title* of the topic (hereafter called original query string) by stripping punctuation and special characters.
  2. build the *Query* by adding the tokens resulting from the application of Lucene's *QueryParser* on the original query string with the specified analyzer, for each field.
  3. build the corresponding boolean queries and add them to the final *Query* with specific boosts dependent on the fields.
- **synonyms queries**:
  1. elimination of entities found by the *NER* python program from the original query string.

[20]https://spacy.io/models/fr

2. analyze the remaining query string via Lucene's *QueryParser* and obtain a set of tokens, which represent all the terms of the original query of which we want to find synonyms.
3. for each token t:
   a) retrieves a specific number of synonyms using the *Word2vec* model.
   b) analyzes each synonym retrieved using the two *QueryParsers* for *body* and *title*.
   c) uses the resulting tokens to build two separate ***SynonymQueries***, one for each field, with the initial content of the token t, by adding all the other tokens with a weight resulting from the following weighting function:

      ```
      model.similarity(t, syn)
      ```

      which calculates the cosine similarity between the synonym and the token t.

      NOTE: we use *SynonymQueries* because, as stated in the official Lucene's documentation:

      > "for scoring purposes, these queries try to score the terms as if you had indexed them as one term: it will match any of the terms but only invoke the similarity a single time, scoring the sum of all term frequencies for the document"[21]

      We found these *SynonymQueries* to be very helpful for improving recall.
   d) add the *SynonymQueries* to the final *Query*, with specific boosts applied depending on the fields.

- **entity queries**:
  1. elimination of the terms to expand, found at the initial step of the process for generating synonym queries, from the original query string.
  2. If the resulting query is not empty, then there must be entity terms. These terms are analyzed using the two *QueryParsers* for *title* and *body*, and the resulting tokens are used to form ***PhraseQueries***[22].

     NOTE: we use *PhraseQueries* because they enable to build queries whose terms depend on the position of each other, such that a query like *"New York"* finds matches in the documents only if the two terms are found being no more distant from each other than a specific value.
  3. Finally, add the two queries to the final *Query* with specific boosts dependent on the fields.

We now definitively build the final *Query* which is passed to the Lucene *IndexSearcher* to perform the actual search and ranking of the documents. Different similarities and analyzers were tested, as reported in Section 5.

Finally, we produce the run.

[21] https://lucene.apache.org/core/8_1_0/core/org/apache/lucene/search/SynonymQuery.html
[22] https://lucene.apache.org/core/6_5_0/core/org/apache/lucene/search/PhraseQuery.html

3.4.4. SearcherW2VRerank

Reranking methods refer to all processes that aim to improve the relevance and effectiveness of the results of an information retrieval system by reordering the documents initially retrieved based on certain criteria. Often reranking criteria can take into account user preferences and other customizations (e.g. browsing history, feedback, etc.), and at times, they may exploit (as in our case) query expansion. This technique consists of expanding the initial queries through additional terms or synonyms and given the broadening of the term spectrum, it can retrieve more relevant documents than the original queries.

We must however consider that, certainly, this expansion of the relevant documents (not returned by the initial query) does lead to an increase in the diversity of results and content, but this not only increases the computational complexity of the system but may also increase the noise of the initial query by distancing itself from the true meaning of the initial query with subsequent loss of information.

We have developed a reranking method based precisely on query expansion in which the key idea behind is a re-ordering of returned documents through the use of:

- synonymous queries
- weighting proportional to the similarity value between the original word and synonym
- rank position of the document in the related query

In detail, the searcher assigns a weight of 2.0 to the original query and a weight equal to the degree of similarity (returned by the previously trained Word2Vec model) between the word (non-entity and non-stopword of the original query) and the new synonym found to each generated synonymous query.

Subsequently, this weight plus the rank position of the relevant document are taken into account in the reranking phase, as it is explained in Section 3.4.4.

Query generation The queries used by this searcher are of two types:

- original query: it is composed of the original term of the input query
- synonym queries: these are constructed by adding the synonym of a word belonging to the original query that is not an entity or a stopword. In particular, the added synonym is the one with the highest similarity value returned by the Word2Vec model described in Section 3.4.3.

Before this implementation, we had tested the direct replacement of the word with its synonym (always replacing it if it was not an entity or a stopword), but the performance dropped, even when we replaced the word with multiple synonyms.

Reranking methodology All queries created by *SearcherW2VRerank* are passed to a reranking function that operates as follows:

1. Identifies the query with the highest weight in the array list of queries passed as input (always the original query in our case).
2. Saves the documents returned by that highest weighted query.
3. Scans all documents returned by all other synonym queries.
4. All documents returned by a query in the input list (different from the original query) that are also present in the documents returned by the query identified in 1. are reranked according to the following formula:

$$\text{final score} = \text{OScore} + \sum_{\text{i}\in\text{all synonym queries}} \left(\text{CScore}_i \cdot \frac{(\text{ maxDocsRetrieved - relativeRankPosition}_i)}{\text{maxDocsRetrieved}} \right) \quad (1)$$

where:

- *OScore* = score of the document in the result list of the highest weighted query (the original query)
- $CScore_i$ = score of the document in the result list of the current query (any synonym queries in our case)
- *maxDocsRetrieved* = maximum number of documents returned per query (length of the list returned by the search method of Lucene's *IndexSearcher*)
- $relativeRankPosition_i$ = position of the document in the list of documents returned by the current query

3.4.5. SearcherAI using ChatGPT's API

Generative Pretrained Transformer (GPT) is a language model developed by OpenAI that can be used for natural language processing tasks such as text generation, summarization, and question-answering. We have used it for query expansion.

The GPT 2 model is publicly available but it is too heavy to use on our local machines. We then opted to use the GPT 3.5 APIs that Open AI makes available.

The idea is simple, using GPT APIs (via com.theokanning.openai java library[23]) we send a prompt to produce a list of n synonyms for a given word in a standard format.

The prompt we used was (asked in french to reduce language errors):
"Énumérez ***n*** synonymes séparés par des point-virgule de: ***query***. Je ne veux pas la définition."
which translates to:

[23]https://github.com/TheoKanning/openai-java

"List ***n*** synonyms separated by semicolon of: ***query***. Do not want the definition".
In this way, we get a list of synonyms separated by a semicolon.
The "Do not want the definition" part was necessary because sometimes ChatGPT replies with a vocabulary definition of the query and not with the synonyms.

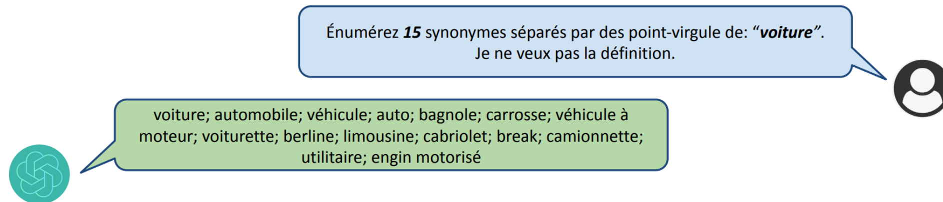


Figure 1: Example of the prompt used

We then find a synonym for each term of the query in a similar way we do in *SearcherW2V* 3.4.3 and add it to the final query as a *SynonymQuery*.

Sometimes ChatGPT does not understand correctly what is asked and could give wrong results, in order to clean up the results we perform on the output these tasks:

- remove non-alphanumeric symbols
- repeat the prompt if the query parser goes in error

The pros of using GPT for query expansion are that it can improve the recall of the system by expanding queries with semantically related terms that may not have been included in the original query. This can help to retrieve more relevant documents that might have been missed by the original query. Also, the neural network can understand typos and "fix them".

The cons are that it can increase the computational cost of the system since it requires additional processing to generate the expanded queries. It can also introduce noise into the system if the expanded queries are not relevant to the user's information needs or if the model makes mistakes. Another problem is that it takes too long to search, requiring every query to make an API request. This would be solved with an offline implementation of GPT3 but it is not yet available to the public.

4. Experimental Setup

In this section, we will describe the overall experimental setup which was used to run the system and produce the runs.

4.1. Data description

The data which was released by CLEF is divided into one training set and two test sets.

Training As stated by the organizers of the task [10],

> "The collection consists of queries and documents provided by the Qwant search Engine[24]. 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 June 2022. In total, the collection contains 672 train queries, with corresponding 9656 assessments coming from the Qwant click model, and 98 heldout queries. The set of documents consists of 1,570,734 downloaded, cleaned, and filtered Web Pages. Apart from their original French versions, the collection also contains translations of the web pages and queries into English. The collection serves as the official training collection for the 2023 LongEval Information Retrieval Lab organized at CLEF."

In addition, URLs to the documents were made available, allowing us to expand upon the indexing and searching capabilities of the system, as shown in 3.

More specifically, we focused our attention on the French part of the dataset, as we understood that documents and, more importantly, queries were translated from Fench, and it did cause a significant drop in the performances of the system when comparing the runs with the qrels for evaluation.

Testing For testing three datasets were used: the one for training, already discussed, and two more testing datasets. These collections were put together following the same process as the training set and they are composed in the following way:

> "The collection contains test datasets for two organized sub-tasks: short-term persistence (sub-task A) and long-term persistence (sub-task B). The data for the short-term persistence sub-task was collected over July 2022 and this dataset contains 1,593,376 documents and 882 queries. The data for the long-term persistence sub-task was collected over September 2022 and this dataset consists of 1,081,334 documents and 923 queries. Apart from the original French versions of the web-pages and queries, the collection also contains their translations into English." [11]

The previous considerations apply to the test sets as well.

[24] https://www.qwant.com

4.2. Evaluation measures

In order to determine the effectiveness of our system and compare our solution we have decided to use four evaluation measures:

- ***Mean Average Precision (MAP)*** is calculated by averaging the precision 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.

$$MAP = \frac{1}{Q}\sum_{q=1}^{Q}\frac{1}{RB}\sum_{k=1}^{K}P(k) \tag{2}$$

  With query q, relevant document k and RB as the total number of relevant documents for the query q.

- ***Normalized Discounted Cumulated Gain (nDCG)*** is a measure that takes into account the relevance of the documents being returned, and normalizes the score based on the ideal ranking order.

$$nDCG@k = \frac{DCG@k}{IDCG@k} \tag{3}$$

- **Recall**: The recall measures how many relevant documents are retrieved by a system out of the total number of relevant documents in the search space. It is calculated as the ratio of relevant documents retrieved to the total number of relevant documents.

$$Recall = \frac{Number\ of\ relevant\ documents\ retrieved}{Number\ of\ relevant\ documents} \tag{4}$$

- **Precision @10**: This is a variant of the precision metric that evaluates the accuracy of a system by considering only the top 10 results. This means that it measures how well the system performs at returning 10 highly relevant documents.

$$Precision@10 = \frac{\sum_{i=1}^{10} ri}{10} \tag{5}$$

4.3. Repository

The repository is organized as follows:

- code: this folder contains the source code of the developed system.
- homework-1: this folder contains the report describing the techniques applied and insights gained.
- homework-2: this folder contains the final paper submitted to CLEF.
- matlab: this folder contains two subfolders, respectively for the Matlab code of the homework-1 and homework-2.

- results: this folder contains the performance scores of the runs.
- runs: this folder contains the runs produced by the developed system.
- slides: this folder contains the slides used for presenting the conducted project.

The repository is available at https://bitbucket.org/upd-dei-stud-prj/seupd2223-squid/src/master/.

4.4. Systems

Different systems were composed by combining together the various components described in Section 3. All systems are based on the French collection. Their performances are shown in Section 5. Among these systems, we have used 5 different ones for producing the runs. These 5 systems are:

- **SQUID_W2V**: it corresponds to the configuration `SearcherW2V, FrenchAnalyzer, BM25`
- **SQUID_BOOST**: it corresponds to the configuration `SearcherBoost, FrenchAnalyzer, BM25`
- **SQUID_W2VRerank**: it corresponds to the configuration `SearcherW2VRerank, FrenchAnalyzer, BM25`
- **SQUID_SEARCHERAI**: it corresponds to the configuration `SearcherAI, CustomAnalyzer, BM25`
- **SQUID_BasicSearcher**: it corresponds to the configuration `BasicSearcher, FrenchAnalyzer, BM25`

4.4.1. Searchers Parameters

Here is how the parameters of the Searchers of the 5 systems have been set to produce the runs:

SearcherW2V & SearcherAI:

- `originalBodyBoost=1.5f` : the boost of the original query to match with the *body* field
- `originalTitleBoost=1.8f` : the boost of the original query to match with the *title* field
- `urlBoost=1.1f` : the boost of the original query to match with the *url* field
- `entityBoostBody=2f` : the boost of the entity query to match with the *body* field
- `entityBoostTitle=3f` : the boost of the entity query to match with the *title* field
- `numSynonyms=15` : the number of synonyms to generate

- `synonymsBoostBody=1f` : the boost of the synonyms query to match with the *body* field

- `synonymsBoostTitle=1f` : the boost of the synonyms query to match with the *title* field

SearcherW2VRerank:

- `bodyBoost=1.5f` : the boost of the original query to match with the *body* field

- `titleBoost=1.8f` : the boost of the original query to match with the *title* field

- `urlBoost=1.1f` : the boost of the original query to match with the *url* field

and its `search` method takes as parameters:

- `pathOfEntitiesFile` : the path of the file that contains all entities of all queries

- `stopListPathFile` : the path of the file that contains all stopwords

- `originalQueryWeight` : the weight of the original query that will be used in document reranking

- `numSynonymsToAdding` : number of synonyms to be added to the query (for each non-entity or stopword term in the original query)

SearcherBoost:

- `boost=IDF`: the value to boost the term in a specific query. The boost is applied only if the term is "rare" in the document collection. The boost is the inverse document frequency of that term in the document collection.

- `DOC_FREQUENCY_THRESHOLD=30` : below this threshold, the term is considered rare, then to boost using IDF.

- `NUM_DOCS=1570734` : number of document in the collection; this value is used to calculate the IDF boost.

BasicSearcher:

- `bodyBoost=1.0f`: the boost of the original query to match with the *body* field

- `titleBoost=1.7f`: the boost of the original query to match with the *title* field

- `urlBoost=1.1f`: the boost of the original query to match with the *url* field

Table 1: systems' performances

Systems	**Anal.**	**Simil.**	**Lang.**	**MAP**	**P@10**	**nDCG**	**R@1000**
SearcherW2V	FA	BM25	fr	**0.2600**	0.1524	**0.4256**	**0.8754**
SearcherW2V	CA	BM25	fr	0.2589	0.1515	0.4211	0.8672
SearcherBoost	FA	BM25	fr	0.2019	0.1223	0.3486	0.7778
SearcherBoost	CA	BM25	fr	0.2010	0.1232	0.3482	0.7800
SearcherBoost	FA	CS	fr	0.2019	0.1223	0.3486	0.7778
SearcherW2VRerank	FA	BM25	fr	0.2506	0.1488	0.4111	0.8443
SearcherAI	CA	BM25	fr	0.2590	**0.1537**	0.4206	0.8639
SearcherAI	FA	BM25	fr	0.2584	0.1530	0.4234	0.8726
BasicSearcher	FA	BM25	fr	0.2519	0.1499	0.4107	0.8466
BM25 Terrier fr reranked by T5	-	BM25	fr	0.2175	0.1329	0.3650	-

5. Results and Discussion

This section shows the results obtained by our systems, on the training and test sets, analyzed using the metrics described in Section 4.2.

5.1. Results on Training data

For discussing the results of the training set, we will take into consideration various systems that were obtained by considering the most interesting combinations of the components described in Section 3.

The systems which were used to produce the runs are 5 among these combinations and have been reported specifically in Section 4.4.

The baseline run we will consider, made available by Prof. Petra Galuščáková, is *BM25 Terrier fr reranked by T5*, which is the best among the ones made available.

We have reported in Table 1 the results of all the systems we tested on the training dataset, including those used to produce the runs, applying the evaluation measures as described in Section 4.2.

In the *system* column we have used the following abbreviations:

- `FA`: French Analyzer
- `CA`: Custom Analyzer
- `BM25`: Best Match 25
- `CS`: Classic Similarity

From the results in Table 1, we can see that the best system we have found is the combination of *SearcherW2V*, `FrenchAnalyzer`, and `BM25Similarity`. With respect to the baseline, we achieved:

- 19.54% improvement in MAP

- 14.67% improvement in P@10
- 16.60% improvement in nDCG

We can see that the MAP score drops using Custom Analyzer: this has led us to stick with the French Analyzer provided by the Lucene library, since the Custom Analyzer stems too much, leading to errors; however, we have seen that in some systems it performs better than the French Analyzer: this is possible thanks to the use of the LetterTokenizer, which separates words until it finds non-letter characters, as defined by the related documentation[25].

All across the board, however, we can denote a general performance boost with `FrenchAnalyzer` and `BM25`.

[25]https://lucene.apache.org/core/7_3_1/analyzers-common/org/apache/lucene/analysis/core/LetterTokenizer.html

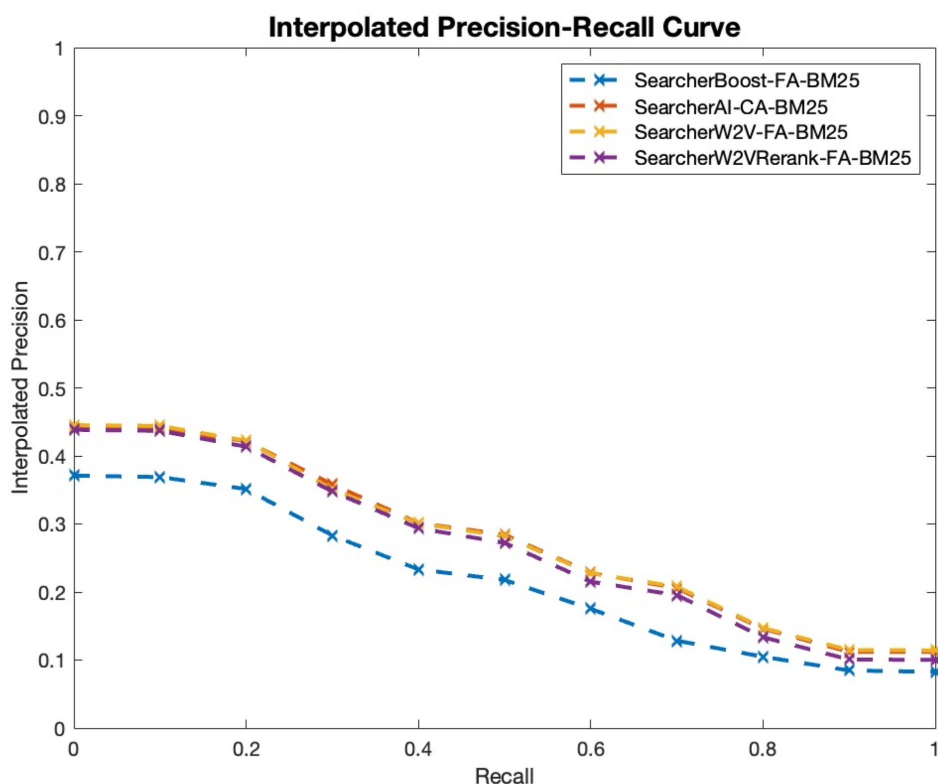


Figure 2: This is the precision-recall curve for the four runs

The precision-recall curve shows that the better systems are the SearcherW2V-FA-BM25 and the SearcherAI-CA-BM25. The curve also shows how the slope is generally low. The slope measures the rate of change of precision for recall and indicates how much improvement the system provides in terms of precision for each additional relevant document retrieved.

5.2. Results on Test data

For discussing the results of the testing data, we will take into consideration only the 5 systems whose runs on the test sets were submitted. More details on these systems can be found in Section 4.4. In particular, as reported in the Table 2, Table 3 and Table 4 the best system in terms of MAP, nDCG and P@10 is now SQUID_SEARCHERAI, while for Recall@1000 SQUID_W2V maintains its superiority.

Furthermore, in light of the findings in the LongEval overview paper [1], it is worth noting that SQUID_SEARCHERAI has achieved top 3 performances on all the test sets. This serves only to stress once again the power of Large Language Models such as GPT 3 in the task of achieving temporal persistence. Extensive training on very large datasets, in fact, allows these models to perform very accurate query expansion, that would otherwise not be possible with

Table 2: Systems performance on the heldout set

Systems	Anal.	Simil.	Lang.	MAP	P@10	nDCG	R@1000
SQUID_BasicSearcher	FA	BM25	fr	0.2522	0.1551	0.4149	0.8408
SQUID_BOOST	FA	BM25	fr	0.2024	0.1255	0.3586	0.7822
SQUID_SEARCHERAI	CA	BM25	fr	**0.2594**	0.1571	**0.4279**	0.8669
SQUID_W2V	FA	BM25	fr	0.2583	**0.1582**	0.4232	**0.8683**
SQUID_W2VRerank	FA	BM25	fr	0.2538	0.1531	0.4154	0.8246

Table 3: Systems performance on the short (July) test set

Systems	Anal.	Simil.	Lang.	MAP	P@10	nDCG	R@1000
SQUID_BasicSearcher	FA	BM25	fr	0.2439	0.1448	0.3998	0.8249
SQUID_BOOST	FA	BM25	fr	0.2248	0.1366	0.3702	0.75
SQUID_SEARCHERAI	CA	BM25	fr	**0.2554**	**0.1494**	**0.4141**	0.8441
SQUID_W2V	FA	BM25	fr	0.2497	0.1486	0.4106	**0.8514**
SQUID_W2VRerank	FA	BM25	fr	0.2440	0.1488	0.3997	0.8208

Table 4: Systems performance on the long (September) test set

Systems	Anal.	Simil.	Lang.	MAP	P@10	nDCG	R@1000
SQUID_BasicSearcher	FA	BM25	fr	0.2425	0.1589	0.411	0.8441
SQUID_BOOST	FA	BM25	fr	0.2175	0.1445	0.374	0.7729
SQUID_SEARCHERAI	CA	BM25	fr	**0.2473**	**0.1627**	**0.4177**	0.8552
SQUID_W2V	FA	BM25	fr	0.2444	0.1611	0.4174	**0.8648**
SQUID_W2VRerank	FA	BM25	fr	0.242	0.1569	0.4105	0.8444

the same proficiency using lexical databases or locally trained neural networks. Additionally, in the latter case, overfitting phenomena are likely to occur, contrarily to LLMs such as GPT 3.

For what concerns the temporal persistence of the submitted systems we can state that the overall performances of the test set collections (heldout, long, term) are just a little bit worse compared to the results of the training data: thus, it is possible to say that the capability of the submitted systems to retrieve and find relevant documents is maintained.

6. Statistical Analysis

In this section, we conduct a statistical analysis of the performances of our systems based on the 5 runs submitted to CLEF. The analysis involves the use of tools such as box plots, two-way ANOVA and Tukey.

Each of the five systems, reported in Section 4.4, has been labeled in Table 5:

Table 5: Systems labeling

Systems	Label
SQUID_SEARCHERAI	1
SQUID_W2V	2
SQUID_BOOST	3
SQUID_W2VRerank	4
SQUID_BasicSearcher	5

6.1. Box Plots

Box plots are graphical tools to represent a distribution of data concisely. In our case, we want to plot the distribution of the scores achieved by our submitted systems on each query of the different test sets, with respect to the evaluation measures of *Average Precision (AP)* and nDCG.

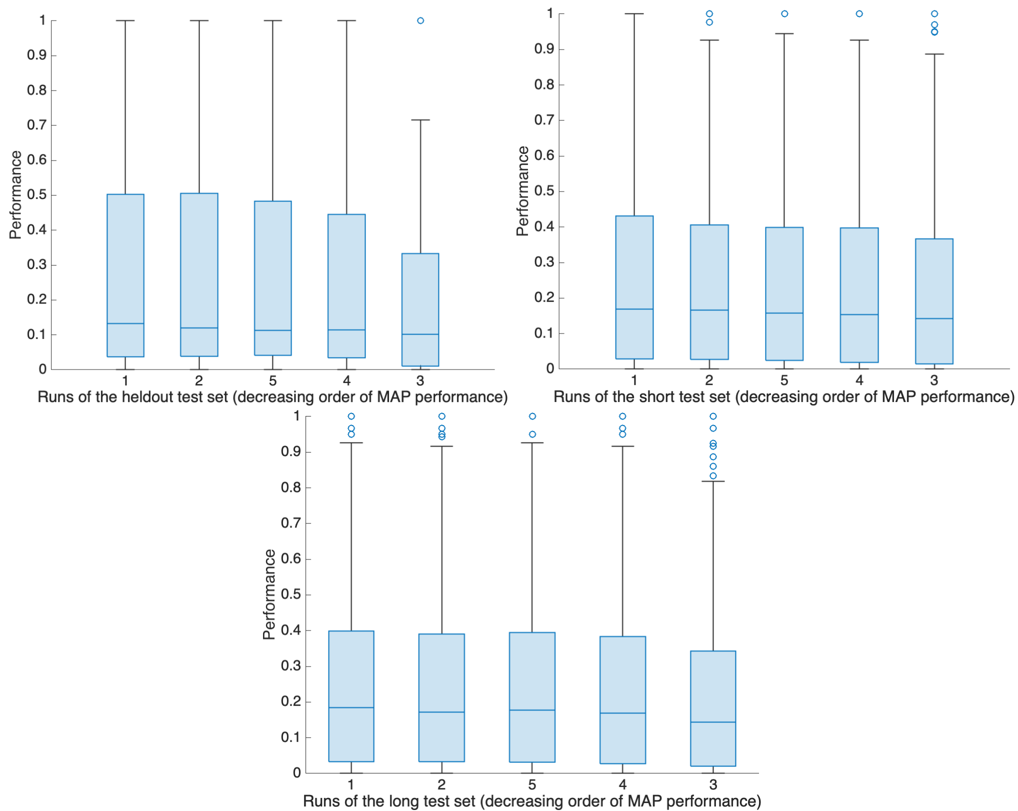


Figure 3: Box plots representations for the AP scores of the five systems on the three test sets

AP By examining the box plots in Figure 3, we can see the medians (which are the positions of the second quartile, representing 50% of the data) are overall on the same level. We can also see that the boxes' heights (which indicate the internal variance between the first and third

quartiles) are comparably similar, as well as the tails of the two whiskers, indicating similar dispersion. The dispersion is overall high, since the range of values spanned by the whiskers and the box are in almost all plots coincident with the entire domain values of the AP measure, meaning that for each query very different results were found overall.

Across the three test sets, we can see that system 3 has a slightly visible shift of the entire box plot towards the lower values of the performance scores compared to the other boxes, indicating somewhat worse performance.

By further examination, we can denote a significant positive (right) skew of the distribution curve in the held-out box plots, indicating high asymmetry since the median value is smaller than the mean, and therefore is in clear contrast to the bell-shaped normal distribution curve. As for the short and long test sets, the skew is not as pronounced, probably due to the higher number of queries, which is able to better stabilize the median value on a more centered position.

Very few outliers are found across all plots, mainly in the system 3 plots. They are all found beyond the tail of the right whisker.

Though we can see that by the ordering of the plots in Figure 3 system 1 has the best mean of the AP (that is MAP) parameter across all test sets, given the analysis provided, we can conclude that no significant difference between the 5 systems are detectable, indicating statistically comparable AP performances.

nDCG With respect to the previous plots, by examining those in Figure 4, we can see those medians are overall on the same level for all systems except system 3, which is lower. The same comments about dispersion previously made are valid in these plots as well.

Across the three test sets, again, we can see that system 3 has a slightly visible shift of the entire box plot towards the lower values of the performance scores compared to the other boxes, indicating somewhat worse performance.

Similar considerations as the previous plots can be made for the skew of the distribution curve.

No outliers are found in these plots.

Though we can see that by the ordering of the plots in Figure 3 system 1 has the best mean of the nDCG parameter across all test sets, given the analysis provided, we can conclude that no significant difference between the 5 systems is detectable, indicating statistically comparable nDCG performances as well.

6.2. Two-way ANOVA

2-way ANOVA belongs to the class of *General Linear Model (GLM)*, it is a statistical analysis technique used to examine the effects of two independent variables on a dependent variable. It assesses whether there are significant differences in the means of the dependent variable across the different levels of each independent variable, as well as any potential interaction between the two independent variables.

In our case, the four independent variables respectively refer to the three test sets of topics and the five submitted systems, while the dependent variable is the output measure: AP or nDCG for our tests.

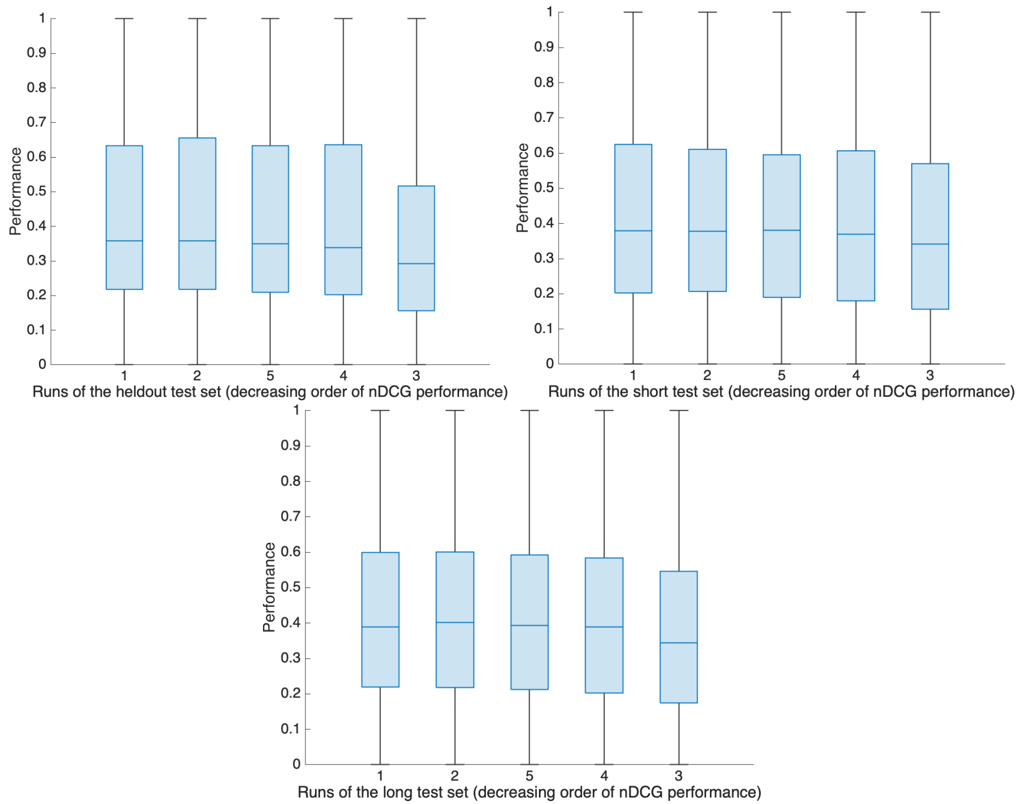


Figure 4: Box plots representations for the nDCG scores of the five systems on the three test sets

By an inspection of the tables related to *Two way Anova*, and in particular on Figure 7 and on Figure 10 the value related to F is really high, and the values of p (Prob>F) are really low, suggesting a high difference among the 5 systems, that requires a deeper analysis by using *Tukey*, discussed on Section 6.3.

Table 6: Anova AP Heldout

Source	SS	df	MS	F	Prob>F
Columns	0.2202	4	0.0550	7.6595	6.0036e-06
Rows	29.9521	97	0.3088	42.9658	2.8503e-159
Error	2.7885	388	0.0072	[]	[]
Total	32.9608	489	[]	[]	[]

Table 7: Anova AP Short

Source	SS	df	MS	F	Prob>F
Columns	0.4724	4	0.1181	13.5258	6.0601e-11
Rows	248.0686	881	0.2816	32.2496	0
Error	30.7686	3524	0.0087	[]	[]
Total	279.3095	4409	[]	[]	[]

Table 8: Anova AP Long

Source	SS	df	MS	F	Prob>F
Columns	0.5266	4	0.1316	6.4170	3.8347e-05
Rows	185.9785	922	0.2017	9.8327	0
Error	75.6571	3688	0.0205	[]	[]
Total	262.1621	4614	[]	[]	[]

Table 9: Anova nDCG Heldout

Source	SS	df	MS	F	Prob>F
Columns	0.3041	4	0.0760	11.6668	5.8875e-09
Rows	29.9589	97	0.3089	47.4015	1.0363e-166
Error	2.5281	388	0.0065	[]	[]
Total	32.7911	489	[]	[]	[]

Table 10: Anova nDCG Short

Source	SS	df	MS	F	Prob>F
Columns	1.0563	4	0.2641	30.6663	3.9516e-25
Rows	270.4425	881	0.3070	35.6474	0
Error	30.3464	3524	0.0086	[]	[]
Total	301.8452	4409	[]	[]	[]

Table 11: Anova nDCG Long

Source	SS	df	MS	F	Prob>F
Columns	1.2106	4	0.3026	13.9120	2.8833e-11
Rows	204.5516	922	0.2219	10.1986	0
Error	80.2277	3688	0.0218	[]	[]
Total	285.9899	4614	[]	[]	[]

6.3. Multiple Comparisons

In order to determine which specific groups differ significantly from each other we have used Tukey's *Honest Significant Difference (HSD)* test: an operation often used subsequently to ANOVA when the F has revealed the existence of a significant difference between some of the tested groups. By an inspection of **AP** of Figure 5 and **nDCG** of Figure 6, we can state that system 3 (in blue) is statistically different from the other 4 systems, which instead result really similar.

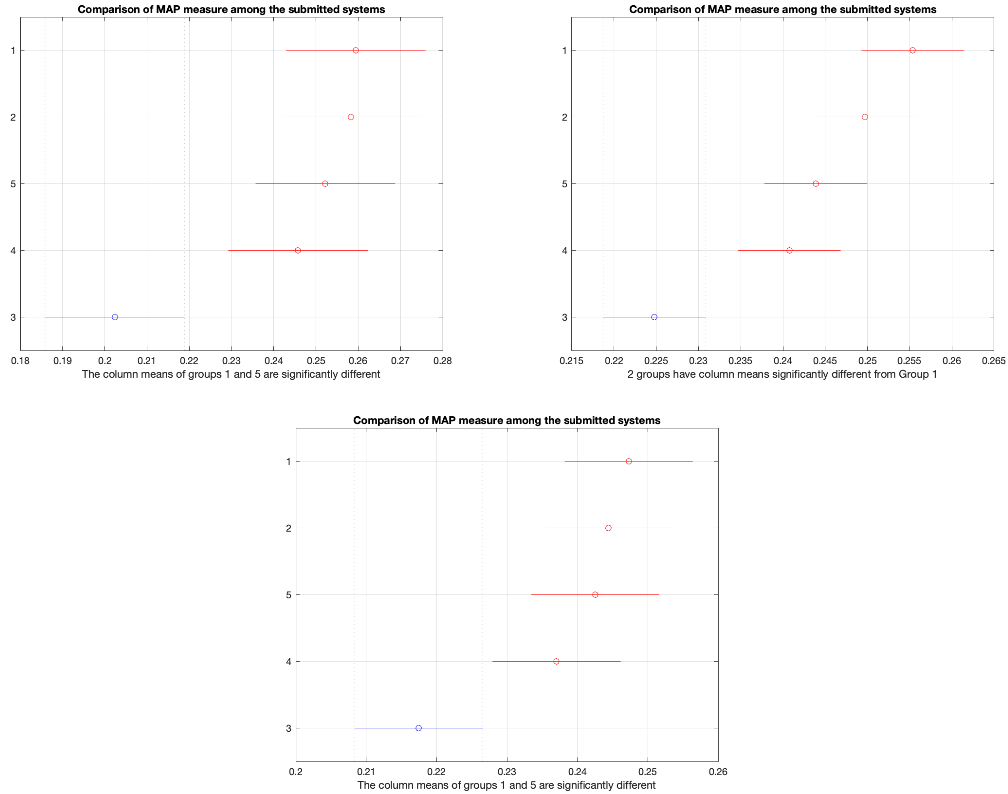


Figure 5: Average Precision

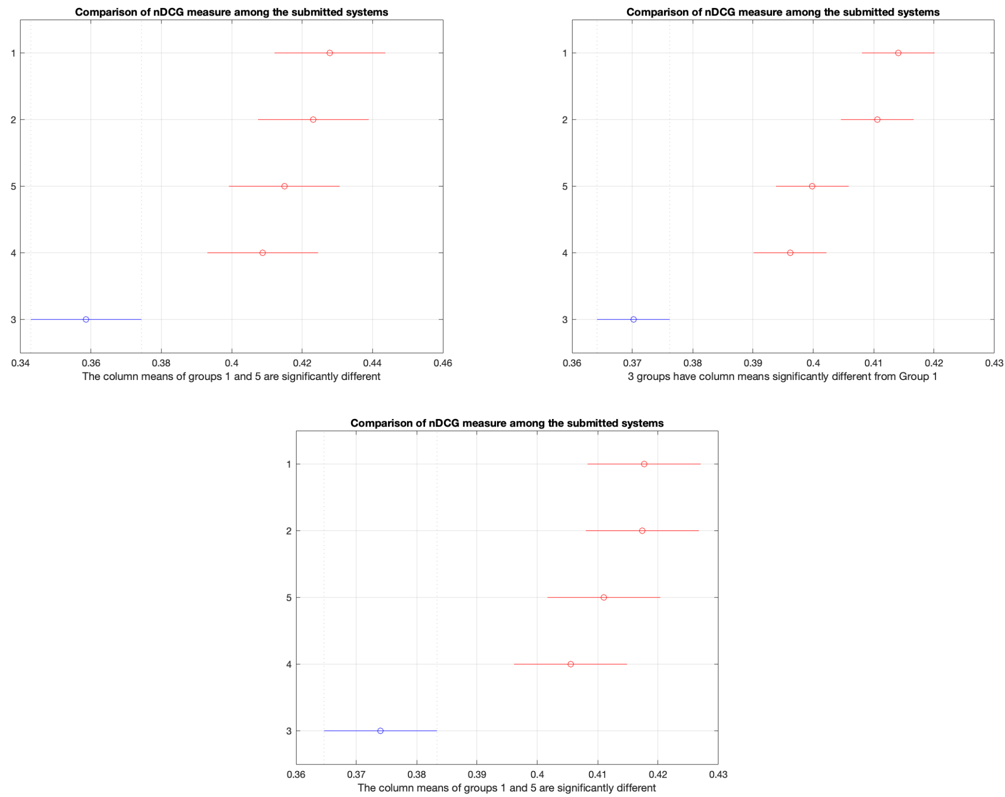


Figure 6: Normalized Discounted Cumulative Gain

7. Conclusions

In conclusion, the goal of maximizing efficiency and effectiveness for our system has led us to the development of a wide variety and combinations of methods, which expanded upon the traditional Information Retrieval systems incorporating different tools from Machine Learning, Deep Learning, and Natural Language Processing fields.

While we were not able to finalize all the features we described, we still managed to put together a well-functioning pipeline, which can be tuned for different retrieval tasks, depending on the required application.

We managed to achieve good improvements with respect to the baseline, with the best system on the training set being, as already reported in Section 5.1, the configuration *SearcherW2V*, `FrenchAnalyzer` and `BM25Similarity`, named SQUID_W2V. On the test sets, however, both SQUID_W2V and SQUID_SEARCHERAI perform equally well, with the latter slightly surpassing the former on precision and nDCG, and vice versa for the other parameters, as reported in Section 5.2.
It is also possible to see from Figure 5 and Figure 6, that all systems except Squid_Boost (the

blue one in the figures) are statistically similar to each other, across all three test sets collections. This shows the promise of the use of Neural Networks and Natural Language Processing tools to achieve state-of-the-art performances in the field of Information Retrieval.

While the systems are able to reach high values in the recall, we cannot really say that the precision, particularly P@10, achieves favorable performances.

We also verified that increasing the number of synonyms in *SearcherW2VRerank* is not helpful in terms of performance, probably due to the poor quality of synonyms generates noise inside the query, creating a query drift effect.

Moreover, for *SearcherW2VRerank*, we verified that changing the stop list does not change the performance so much.

The most promising feature that we believe could help in the precision department is the re-rank using machine learning, whose description is reported in Section 7.1.2, which is one of the features that we were not able to finalize due to our hardware limitations.

Overall, we believe that further work could potentially bring very satisfactory results.

7.1. Future Work

In this section, we want to discuss future improvements to our code and expand upon those features that were not able to finalize or implement.

7.1.1. Word2vec upgrade

While the *Word2vec* implementation did help improving the effectiveness of the system, especially in the recall part, we were not able to fully exploit its potential due to limitations on the hardware at our disposal, as already discussed.

Neural networks do indeed require much computational power for training, and we were only able to train for a very short number of epochs (5), compared to how it is really needed to achieve top-notch performances. More intensive training will most surely improve performance.

Furthermore, were pre-trained models specific for the French language made available, fine-tuning the given collection would be a much faster process than training from scratch, requiring less computational time to achieve a good fit, while also being more robust to changes in the collection itself, thanks to a bigger vocabulary with better-trained weights.

Entity recognition could also be improved, by better training the *Flair* model, as discussed in Section 3.4.3.

7.1.2. Re-ranking

It is possible to improve the system presented by using machine learning for the re-ranking phase. It was possible to verify its effectiveness only partially, due to limited hardware resources.

To simplify the computation, a point-wise approach was opted for, so each object (such as documents) is treated as an independent data point.

A linear regression model had been trained using a total of 32 features (TF, IDF, TF-IDF, DL, BM25, LMIR.ABS, LMIR.DIR and LMIR.JM for the document URL, title, body and the whole document). Given the run output provided by *Lucene*, each pair of queries and documents will be assigned a degree of relevance, and the final ranking will be adjusted. Our implementation of this model generates an accuracy on the test set of 64% but can be highly improved.

LSI algorithm can be introduced in order to speed up the algorithm and improve the robustness. In a nutshell, we create the document-term matrix that describes the occurrences of terms in documents.

The occurrence matrix X is decomposed with *Singular Value Decomposition (SVD)*:

$$X = U\Sigma V \tag{6}$$

We take the *k* largest eigenvalues, which are also the most informative and reduce the matrix. Once we have the "reduced" matrix we can project into the new space the document d and query *q*.

The implementation of reranking algorithm is available in our repository[26] as *LearningToRank.ipynb*. It is also available the code for LSI algorithm as *LSI.py*.

7.1.3. Evolution of GPT

When GPT 3 (or a similar Generative AI) will be available to the open-source community we could use it on our local machine enabling us to summarize automatically the documents we want to index. In this way, we could remove useless information and could search in a faster and more efficient way. At the moment this approach is not feasible because the only way we can use GPT 3 is through the Open AI's API and it requires time and money. Also, with GPT 4 (available only through subscription[27]) we could improve even further thanks to its capability to summarize longer text [28].
Another possible evolution to the system would be to get synonyms of queries from entire phrases rather than singular words, we should study more this approach because of problems caused by non-relevant synonyms that cause lower precision. A possible solution could have

[26]https://bitbucket.org/upd-dei-stud-prj/seupd2223-squid/src/master/
[27]https://openai.com/product/gpt-4
[28]https://theconversation.com/evolution-not-revolution-why-gpt-4-is-notable-but-not-groundbreaking-201858

been to use Named-Entity-Recognition, but it is not applicable because it removes every advantage of creating context-correlated synonyms that GPT provides. With GPT 4 we could generate better synonyms thanks to its better context understanding[29].

References

[1] R. Alkhalifa, I. Bilal, H. Borkakoty, J. Camacho-Collados, R. Deveaud, A. El-Ebshihy, L. Espinosa-Anke, G. Gonzalez-Saez, P. Galuscakova, L. Goeuriot, E. Kochkina, M. Liakata, D. Loureiro, H. T. Madabushi, P. Mulhem, F. Piroi, M. Popel, C. Servan, A. Zubiaga, Overview of the clef-2023 longeval lab on longitudinal evaluation of model performance, in: Experimental IR Meets Multilinguality, Multimodality, and Interaction. Proceedings of the Fourteenth International Conference of the CLEF Association (CLEF 2023), Lecture Notes in Computer Science (LNCS), Springer, Thessaliniki, Greece, 2023.

[2] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, K. Miller, Introduction to wordnet: An on-line lexical database* 3 (1991). doi:10.1093/ijl/3.4.235.

[3] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, 2013. arXiv:1301.3781.

[4] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, R. Vollgraf, FLAIR: An easy-to-use framework for state-of-the-art NLP, in: NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations), 2019, pp. 54–59.

[5] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, 2013. arXiv:1310.4546.

[6] T. Adewumi, F. Liwicki, M. Liwicki, Word2vec: Optimal hyperparameters and their impact on natural language processing downstream tasks, Open Computer Science 12 (2022) 134–141. URL: https://doi.org/10.1515/comp-2022-0236. doi:doi:10.1515/comp-2022-0236.

[7] J. Nivre, M.-C. de Marneffe, F. Ginter, Y. Goldberg, J. Hajič, C. D. Manning, R. McDonald, S. Petrov, S. Pyysalo, N. Silveira, R. Tsarfaty, D. Zeman, Universal Dependencies v1: A multilingual treebank collection, in: Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16), European Language Resources Association (ELRA), Portorož, Slovenia, 2016, pp. 1659–1666. URL: https://aclanthology.org/L16-1262.

[8] A. Akbik, D. Blythe, R. Vollgraf, Contextual string embeddings for sequence labeling, in: COLING 2018, 27th International Conference on Computational Linguistics, 2018, pp. 1638–1649.

[9] P. Koehn, Europarl: A parallel corpus for statistical machine translation, in: Proceedings of Machine Translation Summit X: Papers, Phuket, Thailand, 2005, pp. 79–86. URL: https://aclanthology.org/2005.mtsummit-papers.11.

[10] P. Galuščáková, R. Devaud, G. Gonzalez-Saez, P. Mulhem, L. Goeuriot, F. Piroi, M. Popel, LongEval train collection, 2023. URL: http://hdl.handle.net/11234/1-5010, LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

[29] https://towardsdatascience.com/what-gpt-4-brings-to-the-ai-table-74e392a32ac3

[11] P. Galuščáková, R. Devaud, G. Gonzalez-Saez, P. Mulhem, L. Goeuriot, F. Piroi, M. Popel, LongEval test collection, 2023. URL: http://hdl.handle.net/11234/1-5139, LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.