SEUPD@CLEF Team RISE at LongEval: Improving Search by Crafting Titles and Matching URLs

Notebook for the LongEval Lab at CLEF 2025

Davide Furlan[1], Giulia Gibellato[1], Seyedeh Sara NaziriAlhashem[1], Emanuele Pase[1,*], Alberto Pasqualetto[1], Filippo Tiberio[1] and Nicola Ferro[1]

[1]*University of Padua, Italy*

Abstract

This report outlines the development of the RISE group's Information Retrieval (IR) system for the LongEval-WebRetrieval CLEF 2025 Lab. The objective was to design an efficient, scalable search engine capable of handling large-scale French collections with a focus on consistent performance. The proposed system incorporates a modular architecture, including a parser, an analyzer, an indexer and a searcher, then also query translation and expansion using the Gemini LLM, and a non-neural reranking component to enhance retrieval quality. Emphasis was put on optimizing indexing and searching speed through multi-threading, improving relevance via crafting a title for each document and an URL-based document boosting based on the alignment between user queries and the document's URL. The evaluation has followed a stepwise enhancement approach, beginning with a Lucene-based baseline.

Keywords

Document Parsing, Query Translation, Query Expansion, URL Manipulation, CLEF 2025, LongEval-WebRetrieval, Information Retrieval, Temporal Evolution, Search Engine

1. Introduction

Enormous amounts of data is published every day online: to make it useful it must be stored, parsed, indexed and retrieved efficiently.

The best way to retrieve such data is through the implementation of search engines: a user asks for information (aka does a query) and the model will return a document (a webpage when talking about websites) which satisfies the most the initial request. Search engines play a fundamental role in information retrieval across many different domains such as entertainment, healthcare, education, etc..

Our team's goal is to build an efficient search engine for the CLEF LongEval-WebRetrieval 2025 collection[1]. In our work, we focus on computational speed and scalability while aiming for the best results in terms of the *Normalized Discounted Cumulated Gain (nDCG)* metric.

We will describe the implementation of multiple ideas with the aim of tuning their hyperparameters, comparing, and merging them into a final best-performing system. We decided to analyze the top 5 best-performing systems in terms of nDCG which will be analyzed for their performance robustness over time (the different snapshots of the dataset) and their statistical similarity.

The paper is organized as follows: Section 2 links to LongEval's previous years' related works and the foundational studies that influenced our methodology; Section 3 describes our approach and the pipeline, here we show which are the building blocks of our system; Section 4 describes the experimental setup and links to the codebase; Section 5 discusses our results and the relative statistical tests; finally Section 6 draws some conclusions and outlooks for future work.

CLEF 2025 Working Notes, 9 – 12 September 2025, Madrid, Spain

*Corresponding author.

davide.furlan.10@studenti.unipd.it (D. Furlan); giulia.gibellato@studenti.unipd.it (G. Gibellato); seyedehsara.nazirialhashem@studenti.unipd.it (S. S. NaziriAlhashem); emanuele.pase@studenti.unipd.it (E. Pase); alberto.pasqualetto.2@studenti.unipd.it (A. Pasqualetto); filippo.tiberio@studenti.unipd.it (F. Tiberio); nicola.ferro@unipd.it (N. Ferro)
https://www.dei.unipd.it/~ferro/ (N. Ferro)
0009-0009-9830-7038 (A. Pasqualetto); 0000-0001-9219-6239 (N. Ferro)

2. Related Works

At the beginning of our work, we spent a few days understanding which innovative or interesting solutions our colleagues from previous years implemented. Below are some information we have gathered:

- **JIHUMING team** [2]: They highlighted the avoidance of *Named Entity Recognition (NER)* due to efficiency issues.
- **NEON team** [3]: Avoidance of WordNet usage for query expansion. Not good results because it only works with texts in English.
- **QEVALS team** [4]: They used OpenAI's GPT 3.5 turbo for query expansion. From this idea, we decided to use Google Gemini for query expansion.
- **IRIS team** [5]: From the IRIS team we took the idea of using an additional document field for the URL, with the purpose of saving the web address of the document in case the query explicitly asked for the URL. Additionally, from IRIS we took some ideas about stoplists sets and to use the French Light Stemmer.
- **MOUSE team** [6]: We used the MOUSE team's project, the last year's best performing work coming from UniPD, as a reference of the steps to do and which components can be the best performing ones.

3. Methodology

Apache Lucene is the core framework that we used to build our search engine. It is a high-performance, full-featured text search engine library written in Java. Lucene is widely adopted in the *Information Retrieval (IR)* community. We also implemented Apache OpenNLP for Natural Language Processing tasks, such as tokenization and *Named Entity Recognition (NER)*, but we did not use it in our final system.

3.1. Analyzer

The key component of a search engine built with Apache Lucene is the analyzer. The analyzer processes text data from both documents and queries before indexing and searching, breaking it down into tokens, and applying various transformations or filters to improve the effectiveness of the search.

It is implemented as `RiseAnalyzer`, a Java class that extends the `Analyzer` class from Lucene.

Our implementation of the analyzer is dynamic, which means that it can be changed at runtime without the need to recompile the code. This is achieved by using a JSON configuration file that specifies the components of the analyzer. The configuration file contains a Tokenizer, and a list of `TokenFilters`. Both standard (from Lucene) and OpenNLP analyzers are supported. The following is the configuration file for our best performing analyzer:

Listing 1: Configuration file for the analyzer

```
{
    "tokenizer": {
      "kind": "core",
      "name": "org.apache.lucene.analysis.standard.StandardTokenizer"
    },
    "filters": [
      {
        "kind": "class",
        "name": "org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilter"
      },
      {
        "kind": "class",
            "name": "org.apache.lucene.analysis.LowerCaseFilter"
        },
        {
            "kind": "elision",
            "name": "filters/FR_elision–articles.txt"
        },
        {
            "kind": "class",
            "name": "org.apache.lucene.analysis.fr.FrenchLightStemFilter"
        }
    ]
}
```

This approach allows us to easily experiment with different analyzers and find the one that works best for our data.

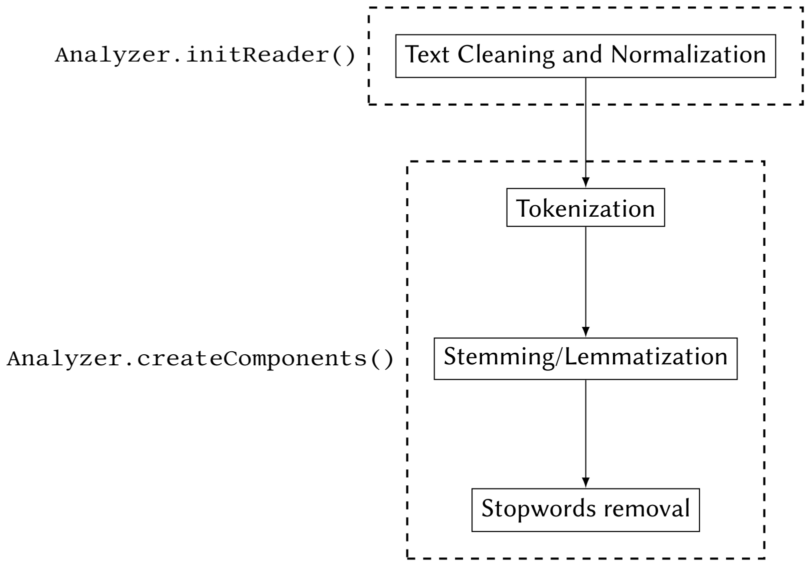


Figure 1: Pipeline of an implementation of `Analyzer` Lucene Class

3.1.1. Text Cleaning and Normalization

Document cleaning is a crucial preprocessing step in IR-based systems. Before tokenization, the pipeline performs extensive normalization and cleaning on raw text, aiming to standardize information such as *dates*, and *phone numbers* to maximize matching with queries presenting dates or phone numbers, while also removing irrelevant symbols, characters, and few-digit numbers.

A modular `TextPreprocessor` Java class has been developed, which performs several transformations:

Date Normalization French and English natural language dates (e.g., *26 janvier 1984*, *January 26th, 1984*) are converted into a unified `dd_mm_yyyy` format using regular expressions and language-specific month mappings. Numeric date formats (e.g. *26 01 1984*) are also normalized to the same structure.

Phone Number Normalization French phone numbers in various formats are standardized to a plain digit format (e.g., `0612345678`) to improve matching and reduce token ambiguity.

Short Number Removal Standalone numbers with 1 to 3 digits that are not part of dates or phone numbers are removed to eliminate noise, which commonly arises from HTML or CSS code coming from the scraping process.

Character Cleanup The pipeline filters out emojis, hashtags, HTML and CSS code, invisible Unicode characters, and styling directives, using a compiled regex pattern.

The cleaning process is integrated into the `RiseAnalyzer` class by overriding the `initReader()` method. This ensures that all text passed through the Lucene pipeline is cleaned consistently and efficiently before tokenization.

One particular challenge was preserving semantically meaningful tokens while eliminating noise: to address this, the cleaning module protects normalized entities (dates and phone numbers) during filtering and restores them afterwards. This multistep protection strategy minimizes false deletions.

By adopting this robust cleaning pipeline, both the quality of the index and the consistency of token-level features are improved, which benefits later stages of query matching.

3.1.2. Tokenization

Tokenization is the fundamental step of any Analyzer and its quality significantly affects the performance. We have explored the following tokenizers with the goal of identifying the most suitable one for our pipeline:

WhitespaceTokenizer (from Lucene) it is a simple tokenizer that splits text on whitespace characters. While fast and lightweight, it lacks the sophistication to handle punctuation or language-specific rules.

LetterTokenizer (from Lucene) it splits tokens at non-letter characters. Although slightly more refined than `WhitespaceTokenizer`, it still falls short in treating acronyms, contractions, and numeric data appropriately.

OpenNLPTokenizer (from OpenNLP) these models give more linguistically-aware tokenizers. They need to be configured with French (or English) sentence and token models. While these tokenizers offer accurate segmentation and sentence-level analysis, they introduce significant runtime overhead, and they focus on just one language.

StandardTokenizer (from Lucene) this is a widely adopted tokenizer in the Lucene framework. It provides robust handling of punctuation, alphanumeric strings, email addresses, acronyms and other linguistic patterns using a grammar-based approach. It balances accuracy and performance effectively without the need for external models.

The latter tokenizer is the one we decided to adopt in our pipeline. This decision was guided not only by our empirical observations and the will of keeping indexing step as lightweight as possible, but also by findings in prior reports (described in section 2). So some of the key reasons behind this choice include proven reliability, balanced trade-offs, and also a sort of community consensus.

3.1.3. Stemming and Lemmatization

The stemmer constitutes the second major step in the offline phase of an IR pipeline. Its primary role is to reduce words to their root forms, enabling the system to match various inflected or derived versions of a word to a common base. This not only helps improve recall by supporting more flexible term matching, but also reduces index size. By collapsing word variants into unified representations, stemming introduces a level of semantic normalization that enhances search effectiveness. An example of stemming is the reduction of the words *change*, *changing*, and *changer* to the root form *chang*. Several stemmers like *PorterStemmer* and *SnowballStemmer* are available in the Lucene library.

A similar tool is the lemmatizer, which also reduces words to their base forms, but does so transforming words to linguistically correct lemmas. For example, a lemmatizer would convert *better* to *good* and *went* to *go*. In Lucene, lemmatizers are implemented through OpenNLP and they need a language model to work.

Lemmatization is more accurate than stemming, but it is also way more computationally expensive. In the context of out project, where we are dealing with a large collection of documents, we opted for stemming to ensure a faster indexing process.

The chosen stemmer is the `FrenchLightStemmer` [7], which is a stemmer distributed with Lucene and is specifically designed for the French language. The choice of the stemmer has also been directed by previous works by Cazzador et al. [6] and Galli et al. [5].

Stoplists Stopwords are very frequent words that do not help to discriminate between documents, and are usually removed from the text prior to indexing.

We tried implementing different stoplists in our indexing process. We tested both stoplists found on the internet and custom stoplists crafted starting from most frequent words in the corpus.

Since some documents are not in French and, even in French passages, English words may appear, we also considered a stoplist for English words. We have been directed to this approach by the previous work of the IRIS team [5].

At first, we used downloaded stoplists to see how they would affect the indexing, but after taking a closer look at the output tokens, we realized that some very common and irrelevant words were still not being filtered out. To fix this, we decided to create some custom stoplists from the most frequent words in the corpus trying to get to a cleaner and smaller index, and an improved search result quality. Most frequent words were retrieved by accessing the index using the Apache Lucene Luke tool.

- **`FR_stoplist.txt`**: a French stoplist found on the internet. It contains 463 words.
- **`customFrenchStoplist.txt`**: a custom stoplist created by us merging the most popular stoplists. It contains 552 words.
- **`SMART.txt`**: the english stoplist of SMART retrieval system [8]. It contains 571 words.
- **`500-custom.txt`**: a custom stoplist created removing the most frequent 500 words in the corpus.
- **`250-custom.txt`**: a custom stoplist created removing the most frequent 250 words in the corpus.
- **`125-custom.txt`**: a custom stoplist created removing the most frequent 125 words in the corpus.
- **`50-custom.txt`**: a custom stoplist created removing the most frequent 50 words in the corpus.

As we will detail in section 3.4.1, the use of stoplists has a negative impact on the quality of the results for this dataset.

Another filter we have implemented, similar to stemming is the `ElisionFilter`, which is a Lucene filter that removes French articles and prepositions with apostrophes from the tokens, in the French language a lot of words are formed by the contraction of a preposition and an article, for example *l'avion* (the plane) will be tokenized as *avion* (plane). To apply such filter we need to provide a list of a few articles to be removed. Those can be found in the `FR_elision-articles.txt` file. We observed this filter being effective, so we implemented it in our baseline.

3.2. Indexer

To enable efficient retrieval and structured storage of documents, we have developed an indexing component implemented in the `RiseIndexer` class.

This class is responsible for orchestrating the entire indexing workflow: reading raw documents from the parsed corpus, analyzing their textual content through the analyzer, and writing the resulting indexable fields.

In our implementation, the indexing logic is designed to be modular, extensible and optimized for performance through multi-threading. The indexer leverages concurrent workers to process and index documents in parallel, significantly reducing the overall indexing time when handling large corpora.

3.2.1. Language Detector

During the development of the search engine, one of the core challenges we have addressed was the support for multilingual documents, specifically in French and English. To this end, we have initially integrated a language detection component using an OpenNLP model. However, this turned out to be a complex and critical aspect of the system design.

Our idea was to automatically detect the language of each document at indexing time and then apply the most appropriate language-specific analysis pipeline (for example a French analyzer for the French documents and an English analyzer for the English ones). This logic has been encapsulated inside a custom `RiseAnalyzerWrapper` class. The wrapper acts as an abstraction layer over multiple analyzers, dynamically choosing the correct one for each document based on the detected language, without requiring manual separation or prior annotation of the documents.

Despite the theoretical value of this approach, we encountered several practical and architectural challenges:

Thread safety and concurrency integrating dynamic analyzer switching into a multi-threaded indexing pipeline introduced complexity and potential synchronization issues;

Performance concerns given the large number of documents, the overhead introduced by on-the-fly language detection was substantial. Indexing time was observed to increase up to threefold, which significantly impacted efficiency.

To better understand the implications of our design choices, we have run experiments using the language detector along with a set of counters (now commented out in the code), used to track the distribution of languages across documents, identify inconsistencies in language detection for documents sharing the same ID and verify detection accuracy relative to expectations. The results of this analysis revealed that the majority of documents are written in French, with only a small portion in English or other languages.

Based on this observation, and given the performance penalties, we concluded that a fully multilingual indexing pipeline was not worthwhile for this particular dataset.

So we ultimately decided to abandon automatic language detection having also the confirmation from the organizers that the task should be French-only, hence non-French documents should be considered noise. Instead, we have opted for a single-language French pipeline. This allows us to maintain high indexing performance and avoid potential inconsistencies in term processing.

On the other hand a language detection and translation component has been applied for the queries (further details about query translation in section 3.3.1).

3.2.2. Parser

In the provided collection documents, split into multiple files, both in JSON and TREC format, are provided. They need to be parsed before they can be indexed. At this stage, we also perform an ad hoc cleaning of the text to remove noise such as HTML tags, CSS code, emojis, and other irrelevant characters.

Since the documents were provided to us in two formats, we conducted analyses on the reading speed of both formats to determine which one would be more efficient to reduce as much as possible the indexing time even from the very first step. To do this, the `ParserBenchmarker` class has been created to compare the reading speed of TREC and JSON documents, both with handcrafted logic (REGEXes), and using specific libraries.

3.2.3. Parsing Speed Testing

Performance evaluations were performed to determine the optimal selection of the parser based on execution speed. To ensure statistical robustness, the parsing times of the complete dataset were recorded across 10 independent runs.

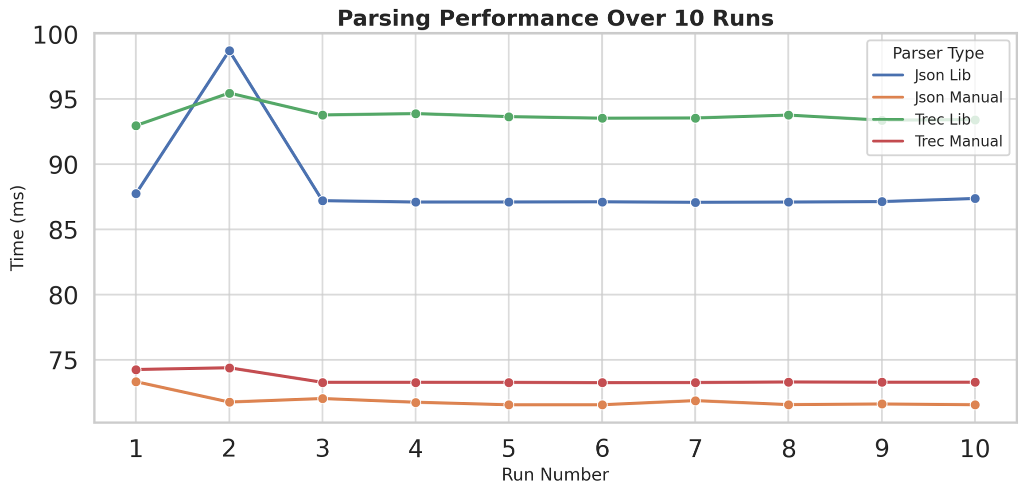


Figure 2: Run/Time plot for the basic parsers

Preliminary results demonstrated that parsers employing a manual implementation approach (using REGEXes), which avoid external JSON/XML parsing libraries, offered significantly superior performance in terms of parsing speed. Subsequently, a statistical comparison between parsing TREC and JSON formatted files was carried out using a two-sided Wilcoxon signed rank test. The results indicated a statistically significant speed advantage when parsing JSON files compared to TREC files ($p\text{-}value = 0.001706$).

To further enhance parsing efficiency, a decision was made to develop a dedicated "one-shot" parser, prioritizing processing speed over memory usage. This parser, also operating on JSON-formatted files, fully loads the dataset into memory utilizing the `com.fasterxml.jackson` library and subsequently returns the parsed objects iteratively via the invocation of the iterator's `next()` method. Although this implementation poses potential risks, such as triggering Out-of-Memory (OOM) errors under insufficient system memory conditions, it provides a substantial speed increase beneficial for indexing processes.

A very important class is `ParsedDocument`, which serves to represent the parsed documents, saving:

- **Id**: The numeric identifier value of the document
- **Title**: Crafted from the first part of the document, as described in paragraph 3.2.3
- **Text**: The body of the document, containing the full text

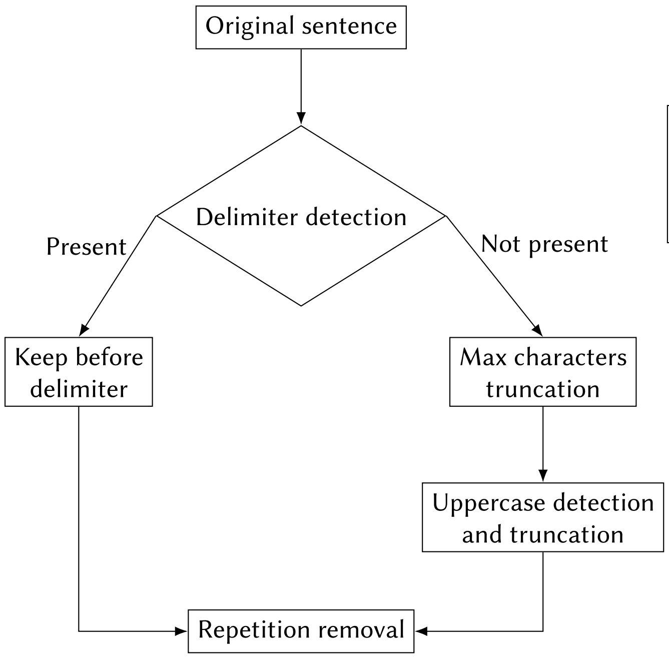
And, if present:

- **URL**: The web address of the document
- **Domain**: The domain of the web address
- **Keyword**: Words present within the URL; used as query expansion to improve document search

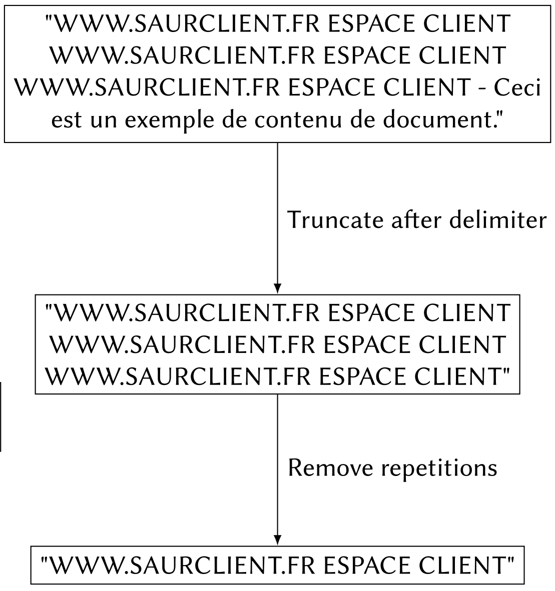
Title extraction Examining the documents in the corpus, we found that frequently the first part of the document contains the title of the webpage.

The title is often a concise and informative summary of the content of the document, which makes it a valuable piece of information for the search engine. Hence we decided to exploit this information and extract the title from the document in order to generate a new field in the index called `title` used to boost the score of the documents during the search phase (see Subsection 3.3.2 for more details).

The algorithm we used is simple: it keeps the part of the document that is before a delimiter (`"|"`, `"-"`, `"?"`, `"!"`, `"."`) and discards the rest. If the title is not found, the algorithm checks if in the beginning of the document there is a substring all in uppercase letters, which is likely to be the title. If the title is not found, the algorithm keeps the first 70 characters of the document as the title.



(a) Pipeline of the title extraction algorithm implemented in the `TitleExtractor` class


"WWW.SAURCLIENT.FR ESPACE CLIENT WWW.SAURCLIENT.FR ESPACE CLIENT WWW.SAURCLIENT.FR ESPACE CLIENT - Ceci est un exemple de contenu de document."
Truncate after delimiter
"WWW.SAURCLIENT.FR ESPACE CLIENT WWW.SAURCLIENT.FR ESPACE CLIENT WWW.SAURCLIENT.FR ESPACE CLIENT"
Remove repetitions
"WWW.SAURCLIENT.FR ESPACE CLIENT"

(b) Example of the title extraction algorithm

Figure 3: Title extraction algorithm

A final processing step is to remove duplicates from the title.

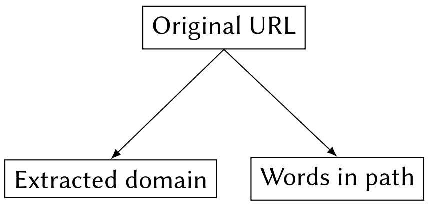
The extraction of the title is done in the `TitleExtractor` class.

URL extraction An important feature we developed is the URL extraction.

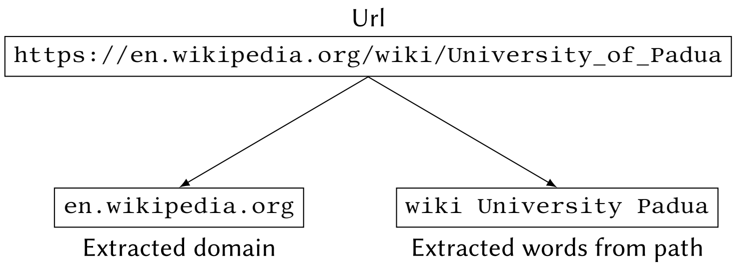
A URL is primarily composed of a domain and a path. This structure is particularly relevant because the domain may appear in the user's query; identifying it allows us to prioritize documents from strongly authoritative sources corresponding to the queried domain. Additionally, the path often contains terms that either reflect an alternative version of the document's title or provide a semantic description of its content. As such, it enables us to boost query tokens that are especially relevant to the document, similarly to how a title does.

The URL extraction occurs simultaneously with the creation of the `ParsedDocument`: When the document ID is available, the system looks in the *release_2025/collection_db.db* database for the entry with the same ID. Then, it processes the URL by extracting the domain and the words present in the address to save them in the respective fields mentioned above.

Here is how URL extraction works in detail:



(a) Pipeline of the URL extraction


Url
https://en.wikipedia.org/wiki/University_of_Padua
en.wikipedia.org
Extracted domain
wiki University Padua
Extracted words from path

(b) Example of the url extraction

Figure 4: URL extraction schema

1. The first step occurs during the creation of the `ParsedDocument`, where the ID of the document is used to seek the corresponding *URL*.
2. If an *URL* is found, it undergoes some manipulation to extract the *domain* and the alphanumeric string after the "/" character following the domain itself.
3. This alphanumeric string is further processed to separate the string into substrings, then numbers, special characters, single letters, and file extensions or properties of the page (like ".pdf", ".html", ".php", ".aspx") are removed. We are then left with words that have a length greater than three characters.
4. The final step is to save the *URL* and the *domain* in their relative fields, and the remaining words appending them to the *title* field of the `ParsedDocument`.

We had a couple of options for where to save the words of the URL (i.e., the strings following the domain name):

Storing them in the document body This option was dismissed because the terms would have been treated as regular content words and likely diluted within the broader context of the document, reducing their influence during retrieval.

Storing them in the document title This alternative appeared more promising, as it allowed us to assign greater weight to the terms, thereby increasing their impact on relevance scoring.

In the end we did not use the domain field since we observed that the url alignment (described in subsection 3.3.3) was already sufficient to boost the documents coming from the same domain.

DB Access Parallelization Query id to URL mappings appear in the *release_2025/collection_db.db* sqlite database present in the task's dataset.

Initially the `ParsingURL` class took into account the whole reading of the database, but we noticed that this was a huge bottleneck in corpus indexing since database queries were sequential, making vain the effort put into the parallelization of indexing step.

In order to overcome this obstacle, we created the `DBReader` class which provides methods to access the database in a fast and parallelizable way, exploiting a read only approach, mainly by disabling the journal usage, disabling the synchronous mode, and optimizing the database which will be accessed in immutable, read-only mode.

Used Pragmas:

```
PRAGMA mmap_size = 268435456
PRAGMA cache_size = -16000
PRAGMA page_size = 8192
PRAGMA read_uncommitted = 1
PRAGMA synchronous = OFF
PRAGMA journal_mode = OFF
PRAGMA analysis_limit = 1000
PRAGMA optimize
```

A `DBReader` class instance, initialized with a database path, will create multiple `Connections` and `PreparedStatements`, one for each thread, and store them in `ConcurrentHashMaps`

This utility class has been also used for topics reading after query translation (described in subsection 3.3.1) and query expansion (described in subsection 3.3.4) has been implemented for parsing topics and assign them one field each.

3.3. Searcher

The Searcher is the last major component of our information retrieval pipeline. It is responsible for retrieving the relevant documents from the indexed set based on a given query. The main components of the searcher are:

Query formulation (`formulateQuery()`) : For each topic, a *BooleanQuery* is created, containing the title and body fields with their respective boosts.

Search execution : This method performs the actual search. `RiseSearcher` is implemented exploiting parallel processing, considerably improving the performance of the system.

The steps involved are:

- For each topic, the Searcher uses the query to retrieve the top documents, ranked by relevance to the query.
- The scores of the retrieved documents are re-ranked.
- The scores are then normalized between 0 and 1 to ensure consistency across queries.
- The results are written to the output file.

Other interesting features of the Searcher are:

- Title boost 3.3.2
- URL Alignment 3.3.3
- Query expansion 3.3.4

3.3.1. French translation of the query

Given that most documents in our dataset are in French, we have implemented a query translation system that converts user queries into French when they are written in another language. This approach helps ensure accurate and consistent retrieval results without requiring a multilingual pipeline.

The translation logic has been implemented in a standalone Python script: `query_translation.py`, this step is performed offline since we exploited the fact we have the full set of queries available in advance inside the `release_2025/queries_db.db` sqlite database.

The pipeline begins by receiving the original query and determining whether translation is necessary. This decision is determined by a prior step: **language detection**.

For this, we used the FAST_LANGDETECT Python library. If the detected language is already French (our target language), the system simply bypasses translation and returns the original query. While, if the query is in another language, the pipeline uses the **translation component**, which interfaces with the Google Gemini 2.0 Flash API.

The API is invoked through a custom HTTP POST request, constructed with a carefully tailored French-language prompt that instructs the model to translate the query literally, without interpretation or additional explanation.

The prompt (in french):

> *Vous êtes un moteur de traduction. Vous recevrez une chaîne de texte arbitraire dans une langue source inconnue.*
>
> *1. Traduisez le texte mot à mot, en préservant exactement tous les termes et la structure.*
>
> *2. Traduisez toujours en français.*
>
> *3. Ne renvoyez que la requête traduite, sans commentaire ni explication.*
>
> *Voici le texte à traduire: <QUERY>*

If translation is successful, the translated query is returned; otherwise, the original query is preserved to ensure no interruption in processing.

This preprocessing pipeline ensures that user queries are consistently aligned with the main language of the indexed documents, improving the accuracy and relevance of retrieval results without introducing significant complexity into the core system.

3.3.2. Title Boost

Since the title field includes both parts of the URL (described in paragraph 3.2.3) and the extracted document title (as described in paragraph 3.2.3), precious information to discriminate relevant documents, we implemented a boost to the terms present in the title field. This boost is applied during the search phase, where the title terms are given a higher weight compared to terms which appear only in the body.

3.3.3. URL Alignment

It was decided to perform a first re-ranking of the documents using the url of the documents: An alignment between the URL and the query was chosen to assess the amount of overlap between the two. Specifically, we decided to use a normalized global alignment parameterized to $+1$ for matches and -1 for insertions/deletions. In practice, the version based on the dynamic programming technique of the Needleman-Wunsh algorithm has been used: this algorithm calculates the alignment score and normalizes it in the range $[-1, +1]$. The only exception results from the fact that: if the query is an exact substring of the URL, then the score is 1 regardless of the alignment.

```
Example of our alignment:
original url: code.google.com/archive/p/stop-words/
original query: code google.com stopwords

Aligned url:   code.-google.com/archive/p/stop-words/
Aligned query: code- google.com-----------stop-words-
Score:         ++++--++++++++++-----------++++-+++++-
```

In order to also parameterize the boost factor $[min_{boost}, max_{boost}]$ during reranking, the score is interpolated by a normalized sigmoidal function centered on zero: the score is first compressed into the range $[0, 1]$, the normalized sigmoidal function is computed, and then extended to the boost factor range. As in the following:

$$t = \frac{score + 1}{2}, \quad k = 10$$

$$f(t) = \frac{1}{1 + e^{-k(t-0.5)}}, \quad s_0 = \frac{1}{1 + e^{k \cdot 0.5}}, \quad s_1 = \frac{1}{1 + e^{-k \cdot 0.5}}$$

$$f_{\text{norm}}(t) = \frac{f(t) - s_0}{s_1 - s_0}$$

$$f_{\text{final}}(t) = min_{boost} + f_{\text{norm}}(t) \cdot (max_{boost} - min_{boost})$$

Essentially, this approach aims to leverage the semantic content of the URL in order to reward documents with strong alignment and, possibly, slightly penalize those with poor matching.

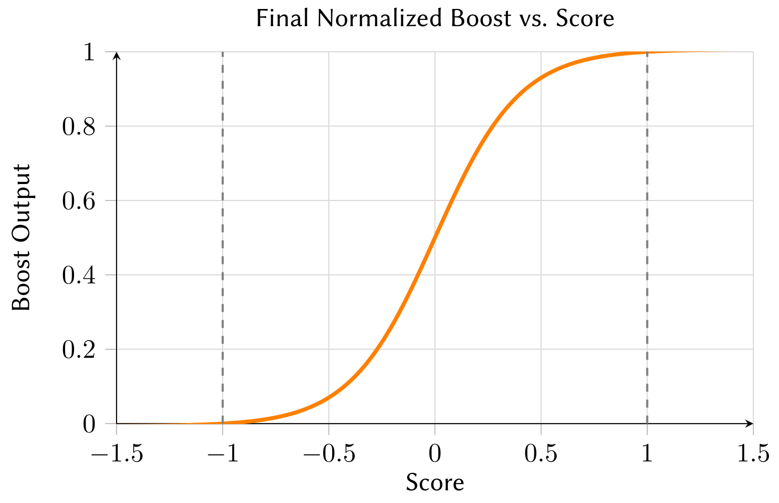


Figure 5: Example of our sigmoid interpolation function for url boosting. Here $k = 10$, $min_{boost} = 0.8$ and $max_{boost} = 1.4$

3.3.4. Query Expansion

We query expand the queries through Google Gemini 2.0 Flash: it permits us to get synonyms and related concepts to the query in analysis. Our prompt to Gemini gives some examples to the model to gain the benefits of few-shot prompting compared to zero-shot one:

Nevertheless, we asked Gemini to answer using structured output, so we can easily parse and use it in our pipeline.

The prompt (in french):

> *Vous faites partie d'un système d'information qui traite les requêtes des utilisateurs. Vous développez une requête donnée en requêtes de sens similaire.*
>
> *Développez la requête de recherche suivante en une liste de mots-clés et expressions associés. Fournissez la sortie sous forme de tableau JSON de chaînes de caractères. N'incluez aucun autre texte en dehors du tableau JSON.*
>
> *Exemples :*
>
> *Requête : recettes de cuisine faciles ["recettes rapides", "cuisine simple", "idées repas faciles", "préparer à manger rapidement", "recettes pour débutants", "meilleures recettes faciles à faire"]*
>
> *Requête : apprendre le français en ligne ["cours de français en ligne", "sites pour apprendre le français", "applications pour le français", "exercices de français en ligne", "apprentissage du français à distance"]*
>
> *Requête : <QUERY>*

Of particular concern are some queries related to sexual content and violence since they won't be query-expanded even with safety filters turned off. If for this or any other reason the model doesn't return any output, we won't consider any expansion for the query.

3.3.5. Reranking

Re-ranking refers to the process of re-evaluating and adjusting the scores of documents retrieved by a first-stage retriever (BM25) by computing similarity between the query and candidate documents using their semantic meaning with the goal of improving retrieval quality.

In our system, we explored multiple reranking strategies during development. Apart from the **manual URL alignment-based re-scoring** method detailed in section 3.3.3, some **bi-encoders** and **cross-encoders** have been tested to try achieving a higher nDCG score.

Tested bi-encoder: *paraphrase-multilingual-MiniLM-L12-v2*.

Tested cross-encoder: *mixedbread-ai/mxbai-rerank-base-v2*.

Dense neural models are computationally expensive, for this reason they are used only for reranking on a few documents.

During our exploration in the training set we tested the previously mentioned rerankers, but we observed that the outcome was not satisfying, sometimes not even improving while paying a huge cost in latency, even with high-performance hardware (NVIDIA H200 GPU) considering on average 10000 queries per month.

For this reason we decided to cut out this step from our pipeline.

> We need to precise that we obtained such results using qrels that were afterwards modified: at the time of our tests the qrels had some relevance judgments that were clearly incorrect. Given the higher focus on semantics of neural IR models with respect to BM25, such inaccurate qrels may have influenced the deterioration of performance we noticed.
>
> Due to computational constraints, we were unable to re-run the reranking on the training and test set.

3.4. Parameters Optimization

We implemented the previously cited features block by block into the pipeline and evaluated the contribution of each individual block when concatenated with the preceding ones. For each block, an optimization phase was carried out to identify the optimal parameters.

3.4.1. Baseline

Our initial baseline, which already yielded satisfactory results has been fixed after testing several analyzer combinations with the components (tokenizer, stoplists, etc....) detailed in section 3.

As a next step, we implemented stopword removal in the pipeline, initially using publicly available French and English stoplists containing approximately 500 words each. However, we observed a decrease in the nDCG score compared to the baseline. To investigate further, we created custom stopword lists composed of the top-k most frequent indexed terms. We found that performance was sensitive to the size of the stoplists: reducing their size led to improved results, with nDCG values eventually returning to baseline levels.

Therefore, we decided to skip the stopword removal step and define as BASELINE the "vanilla" Lucene pipeline, which employs an analyzer composed of the following components: `org.apache.lucene.analysis.standard.StandardTokenizer +`

- `org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilter`
- `org.apache.lucene.analysis.LowerCaseFilter`
- `FR_elision-articles.txt`
- `org.apache.lucene.analysis.fr.FrenchLightStemFilter`

With this setup, we obtained the initial nDCG values in Table 1.

Table 1
nDCG values for BASELINE System

	2022-06	2022-07	2022-08	2022-09	2022-10	2022-11	2022-12	2023-01	2023-02
BASELINE	0.3650	0.3702	0.3772	0.3750	0.4248	0.4267	0.4366	0.4409	0.4381

3.5. Grid Optimization

During our tests, as explained before, we implemented step by step each "block", hence we found the approximate hyperparameters/boosting factors that most enhanced the nDCG in the training set.

Having these values, we performed a grid optimization in multiple finer-coarse steps to fine-tune them and find the hyperparameters that best suited for our dataset.

Such optimization has been performed on only one month since we observed that all systems tend to have the same performance enhancement trend on all months: a system better than another in a month is better than it also in the other months.

First, we conducted the search experiments only with title boost, query expansion, and query translation enabled, while disabling URL alignment (figure 8). These features were tested together but separately from the others, as they interact with one another due to their shared reliance on the document's body field.

Once we obtained the 5 best configurations out of the search, we determined the best hyperparameters for url alignment as second step (figure 9).

At this point, the five best parameter combinations were identified, and the following different systems were also added as reference to the comparison:

- The baseline: no optimization applied, i.e., all parameters set to zero
- The baseline with URL boost, without title-related boosts
- The baseline with title-related boosts, without URL boost

In Table 2, the selected systems are presented, while Table 4 reports the results obtained. Only the 5 best systems have been sent for evaluation to LongEval.

A legend of acronyms can be found in Table 3.

Table 2
Selected systems after grid optimization of parameters + references

1_s_tb0.0_qe0.0_qt0.0_l0.0_u0.0 }	⇛ *(Baseline)*
2_s_tb0.0_qe0.0_qt0.0_l0.7_u1.2 }	⇛ *(Baseline + URL-Boost)*
3_s_tb0.5_qe1.0_qt0.0_l0.0_u0.0 }	⇛ *(Baseline + Title-Boosts)*
4a_s_tb0.5_qe0.1_qt0.1_l0.7_u1.2	
4b_s_tb0.5_qe0.1_qt0.1_l0.8_u1.3	
4c_s_tb0.5_qe0.1_qt0.0_l0.8_u1.3	⇛ *(5 best systems)*
4d_s_tb0.5_qe0.1_qt0.05_l0.8_u1.3	
4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2	

Table 3
Legend of system configuration components. Each system ID is composed of several parameter settings

Component	**Description**
`s`	System identifier (fixed prefix)
`tbX.X`	Title boost weight
`qeX.X`	Query expansion boost weight
`qtX.X`	Query translation boost weight
`lX.X`	URL alignment min_{boost}
`uX.X`	URL alignment max_{boost}

Table 4
nDCG values for each optimal block configuration († on systems not submitted to LongEval)

	2022-06	2022-07	2022-08	2022-09	2022-10	2022-11	2022-12	2023-01	2023-02
1_s_tb0.0_qe0.0_qt0.0_l0.0_u0.0 †	0.3650	0.3702	0.3772	0.3750	0.4248	0.4267	0.4366	0.4409	0.4381
2_s_tb0.0_qe0.0_qt0.0_l0.7_u1.2 †	0.4021	0.4074	0.4105	0.4007	0.4482	0.4527	0.4615	0.4668	0.4737
3_s_tb0.5_qe0.1_qt0.0_l0.0_u0.0 †	0.4389	0.4415	0.4448	0.4436	0.4905	0.4894	0.4932	0.5037	0.5018
4a_s_tb0.5_qe0.1_qt0.1_l0.7_u1.2	0.4788	0.4810	0.4797	0.4718	0.5119	0.5143	0.5178	0.5278	0.5343
4b_s_tb0.5_qe0.1_qt0.1_l0.8_u1.3	0.4785	0.4807	0.4803	0.4718	0.5127	0.5144	0.5182	0.5284	0.5345
4c_s_tb0.5_qe0.1_qt0.0_l0.8_u1.3	**0.4786**	**0.4807**	**0.4801**	**0.4719**	**0.5127**	**0.5144**	**0.5182**	**0.5284**	**0.5346**
4d_s_tb0.5_qe0.1_qt0.05_l0.8_u1.3	0.4785	0.4807	0.4801	0.4718	0.5128	0.5144	0.5181	0.5284	0.5345
4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2	0.4790	0.4811	0.4796	0.4722	0.5121	0.5143	0.5179	0.5279	0.5344

4. Experimental Setup

4.1. Used Collections

For the experiments, we employed the training collection TrecEval25 provided by TrecEval. The collection is organized into 9 separate folders, each containing JSON and TREC files. Each folder was treated as an independent dataset - here the longitudinal evaluation that gives the name to the task. Additionally, the provided database files (.db) was utilized to extract the corresponding URLs for each document and to perform query translation and expansion.

4.2. Evaluation Measures

Performance evaluation throughout this work was conducted using the nDCG metric.

$$\text{nDCG(k)} = \frac{\text{DCG(k)}}{\text{iDCG(k)}} \tag{1}$$

where *Discounted Cumulated Gain (DCG))* is computed as:

$$\text{DCG(k)} = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{2}$$

with rel_i representing the graded relevance of the item at position i, and *Ideal Discounted Cumulated Gain (iDCG)* is the DCG(k) value obtained by sorting documents in decreasing order of relevance, thus representing the optimal ranking.

The metrics were extracted from the runs using the *trec_eval* tool (https://github.com/usnistgov/trec_eval) using the flag -c which considers in the evaluation not only the intersection between topics and qrels, but all the queries in the relevance judgments.

The qrels considered in this work are only those with at least one relevant document.

We also considered the *Relative nDCG Drop (RnD)*: the other evaluation measure considered by LongEval, which is measured by computing the difference between snapshots test sets. This measure supports the evaluation of the impact of the data changes on the systems' results. An analysis on performance trends over time through RnD is presented in Section 5.2.

4.3. Git Repository

The entire project, including the source code, generated outputs, analysis, and additional documentation, is available on Bitbucket at the following URL: https://bitbucket.org/upd-dei-stud-prj/seupd2425-rise/src/master/

4.4. Hardware Used for Experiments

The experiments were conducted on a Windows 10 PC configured as follows:

- CPU: Intel Core i5-10400F
- RAM: 32 GB DDR4 at 3600 MHz
- GPU: NVIDIA GTX 1660
- Storage: Dedicated Samsung 990 EVO NVMe SSD

5. Results

In Table 5 are the performances of our systems in the test set.

Table 5
nDCG performances over test set ($*$ on statistically different systems from the best one, † on systems not submitted to LongEval)

	2023-03	2023-04	2023-05	2023-06	2023-07	2023-08
1_s_tb0.0_qe0.0_qt0.0_l0.0_u0.0 $*$ †	0.4361	0.4533	0.4512	0.4718	0.4347	0.3616
2_s_tb0.0_qe0.0_qt0.0_l0.7_u1.2 $*$ †	0.4769	0.4808	0.4831	0.4988	0.4668	0.3818
3_s_tb0.5_qe0.1_qt0.0_l0.0_u0.0 $*$ †	0.5049	0.5115	0.5140	0.5319	0.4960	0.4052
4a_s_tb0.5_qe0.1_qt0.1_l0.7_u1.2	0.5419	0.5361	0.5430	0.5589	0.5258	0.4214
4b_s_tb0.5_qe0.1_qt0.1_l0.8_u1.3	0.5412	0.5366	0.5430	0.5590	0.5256	0.4219
4c_s_tb0.5_qe0.1_qt0.0_l0.8_u1.3	0.5413	0.5367	0.5429	0.5589	0.5257	0.4218
4d_s_tb0.5_qe0.1_qt0.05_l0.8_u1.3	0.5413	0.5368	0.5430	0.5590	0.5256	0.4217
4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2	**0.5420**	**0.5362**	**0.5430**	**0.5591**	**0.5258**	**0.4213**

In the following section, we validate the significance of improvements introduced in our pipeline using a statistical approach. We focus on the nDCG metric across multiple queries and configurations. Specifically, we apply Two-Way ANOVA followed by Tukey's *Honestly Significant Difference (HSD)* test given as significance level $\alpha = 0.05$.

5.1. Hypothesis Testing

The Two-Way ANOVA followed by Tukey's HSD test revealed several clear and insightful trends in the performance of our retrieval systems: the results confirm that the differences between system configurations are statistically significant when comparing the baseline to systems enhanced with query expansion and title boosting. The baseline configuration, *1_s_tb0.0_qe0.0_qt0.0_l0.0_u0.0*, consistently underperforms relatively to all the other configurations. For example, its comparison with *4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2* yields a mean difference of 0.10588 with a p-value < 0.001, indicating a highly significant improvement. Similar results were observed against nearly all enhanced systems, all with p-values effectively at zero. This strongly supports the conclusion that adding query expansion and title boosting contributes meaningfully to retrieval effectiveness.

Moreover, even switching from l=0.0, u=0.0 to l=0.7, u=1.2 in the baseline system yields a statistically significant improvement, indicating that, even without query expansion, in the baseline system these changes can also provide measurable gains.

Instead, when comparing systems within the same "family" of configurations (i.e., all using the same multipliers for title boosting and query expansion, like tb=0.5 and qe=0.1) we observe that differences due to fine-grained changes are not statistically significant. For example, the system *4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2* compared with the system *4a_s_tb0.5_qe0.1_qt0.1_l0.7_u1.2* yields a mean difference of only 0.000041, with a p-value = 1.00, showing no evidence of a meaningful performance gap. In general, all such within-family comparisons exhibit mean differences < 0.001, with p-values of 1.00, indicating complete statistical overlap.

These considerations suggest that most of the performance gains stem from the presence or absence of key features such as query expansion and title boosting. In figure 6 it is possible to see the ANOVA results.

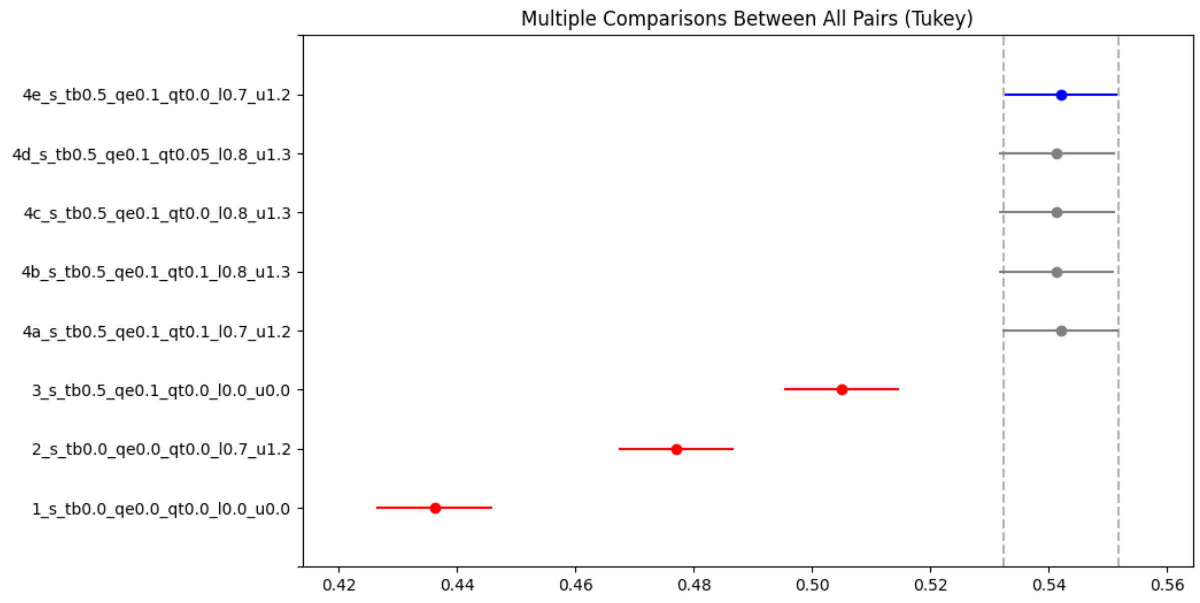


Figure 6: 2-way ANOVA with Tukey HSD test system comparison on March 2023 dataset split: it is clearly visible that 5 systems out of 8 can be considered statistically equivalent. The other months have the same behavior

The statistical evaluation is extended to the different months of the collection and their behavior is the same. We implemented the code to evaluate 3-way ANOVA, but we did not have enough time to fully execute it and report the results. A further comparison on different months split is brought in the next section.

5.2. Long vs Short term behavior

To measure the systems' temporal robustness we exploited the *Relative nDCG Drop (RnD)* measure for each month, by computing the nDCG difference between months as described by the LongEval evaluation guidelines.

We can consider the training set part as *short term behavior* and the test set part as *long term behavior*.

The results are reported in Table 6 and Figure 7.

From the plot in figure 7, it can be observed that the system's performance increase until June 2023, after which it begins a rapid decline until the end of the observed period. With so few months of data, it is difficult to assess the consistency of this trend or determine its underlying causes.

We also claim that our systems' performances are unlinked from time since in long term (test set after 12 months) we see the highest peak in nDCG performances and also the greatest drop (test set after 14 months), while in the very short term the system has stable performances. We hypothesize that these changes in performances of all systems together are primarily due to queries, documents, or relevant judgments quality or updates of their creation system.

An interesting observation is that more complex systems, the ones that perform the best in absolute value, exhibit smaller performance gains and larger degradations compared to simpler systems, including the baseline. This suggests that our fine-tuned models may be overfitting to the specific dataset.

Table 6
Relative nDCG Drop, values between parentheses are relative to 2022-06, values without parentheses are relative to 2023-03 (test set) (∗ on statistically different systems from the best one; † on systems not submitted to LongEval)

	2022-06	2022-07	2022-08	2022-09	2022-10	2022-11	2022-12	2023-01	2023-02	2023-03	2023-04	2023-05	2023-06	2023-07	2023-08
1_s_tb0.0_qe0.0_qt0.0_l0.0_u0.0 ∗ †	(0.0)	(0.0052)	(0.0122)	(0.0100)	(0.0598)	(0.0617)	(0.0716)	(0.0759)	(0.0731)	(0.0711)	(0.0883)	(0.0862)	(0.1068)	(0.0697)	(-0.0034)
										0.0	0.0172	0.0151	0.0357	-0.0014	-0.0745
2_s_tb0.0_qe0.0_qt0.0_l0.7_u1.2 ∗ †	(0.0)	(0.0053)	(0.0084)	(-0.0014)	(0.0461)	(0.0506)	(0.0594)	(0.0647)	(0.0716)	(0.0748)	(0.0787)	(0.0810)	(0.0967)	(0.0647)	(-0.0203)
										0.0	0.0039	0.0062	0.0219	-0.0101	-0.0951
3_s_tb0.5_qe0.1_qt0.0_l0.0_u0.0 ∗ †	(0.0)	(0.0026)	(0.0059)	(0.0047)	(0.0516)	(0.0505)	(0.0543)	(0.0648)	(0.0629)	(0.0660)	(0.0726)	(0.0751)	(0.0930)	(0.0571)	(-0.0337)
										0.0	0.0066	0.0091	0.0270	-0.0089	-0.0997
4a_s_tb0.5_qe0.1_qt0.1_l0.7_u1.2	(0.0)	(0.0022)	(0.0009)	(-0.0070)	(0.0331)	(0.0355)	(0.0390)	(0.0490)	(0.0555)	(0.0631)	(0.0573)	(0.0642)	(0.0801)	(0.0470)	(-0.0574)
										0.0	-0.0058	0.0011	0.0170	-0.0161	-0.1205
4b_s_tb0.5_qe0.1_qt0.1_l0.8_u1.3	(0.0)	(0.0022)	(0.0018)	(-0.0067)	(0.0342)	(0.0359)	(0.0397)	(0.0499)	(0.0560)	(0.0627)	(0.0581)	(0.0645)	(0.0805)	(0.0471)	(-0.0566)
										0.0	-0.0046	0.0018	0.0178	-0.0156	-0.1193
4c_s_tb0.5_qe0.1_qt0.0_l0.8_u1.3	(0.0)	(0.0021)	(0.0015)	(-0.0067)	(0.0341)	(0.0358)	(0.0396)	(0.0498)	(0.0560)	(0.0627)	(0.0581)	(0.0643)	(0.0803)	(0.0471)	(-0.0568)
										0.0	-0.0046	0.0016	0.0176	-0.0156	-0.1195
4d_s_tb0.5_qe0.1_qt0.05_l0.8_u1.3	(0.0)	(0.0022)	(0.0016)	(-0.0067)	(0.0343)	(0.0359)	(0.0396)	(0.0499)	(0.0560)	(0.0628)	(0.0583)	(0.0645)	(0.0805)	(0.0471)	(-0.0568)
										0.0	-0.0045	0.0017	0.0177	-0.0157	-0.1196
4e_s_tb0.5_qe0.1_qt0.0_l0.7_u1.2	(0.0)	(0.0021)	(0.0006)	(-0.0068)	(0.0331)	(0.0353)	(0.0389)	(0.0489)	(0.0554)	(0.0630)	(0.0572)	(0.0640)	(0.0801)	(0.0468)	(-0.0577)
										0.0	**-0.0058**	**0.0010**	**0.0171**	**-0.0162**	**-0.1207**

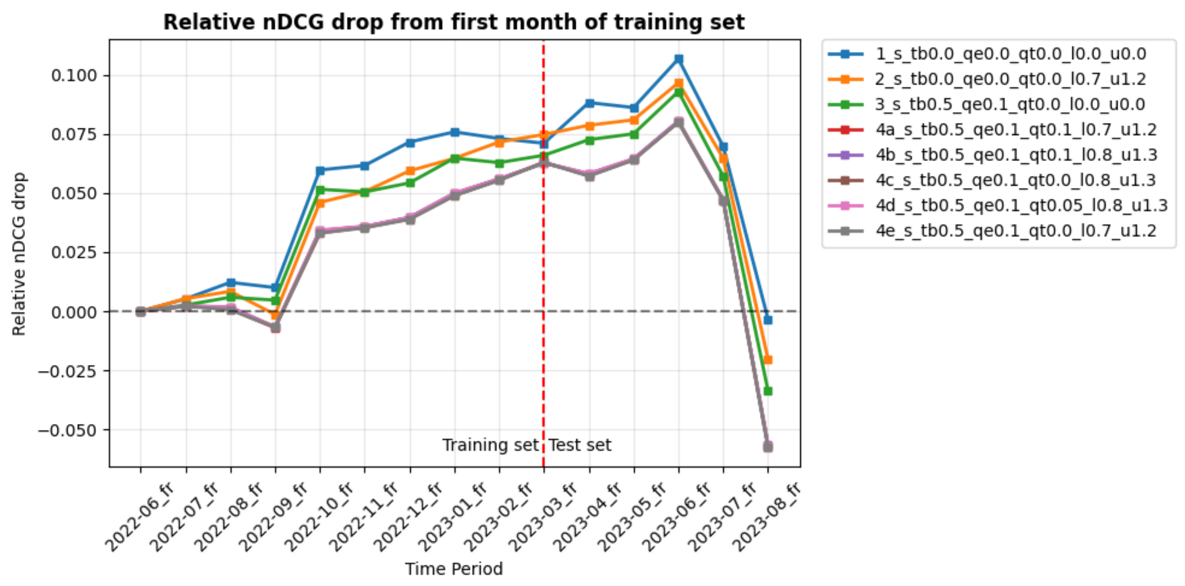


Figure 7: Relative nDCG Drop over train and test set

6. Conclusions and Future Work

We began our project by creating a basic working structure, referred to as the baseline, which implemented a simple version of our pipeline: Parser–Analyzer–Indexer–Searcher. From this baseline, we iteratively introduced and tested various enhancements to improve performance. These included integrating custom stopwords, implementing title boosting, applying URL alignment boosting, expanding queries using Google Gemini, and reranking results. Each component and configuration was evaluated using the *nDCG* metric to determine its impact.

The results are promising and are mainly pushed by the title-boosting factor and secondarily by leveraging the URL information.

Our system appears to be independent of temporal factors, as its performance does not exhibit a clear stable, increasing, or decreasing trend over time, but, while obtaining best-in-class nDCG performances, we notice an overfit on task's data.

Future efforts could focus on refining query strategies and exploring additional optimization techniques, such as:

- Further explore the usage of URL: the words extracted from the url are not currently used, but they may result in another increase of performance comparable to the one already obtained by query expansion.
- Trying to use n-grams: by incorporating n-grams, the system can capture contextual relationships between consecutive words rather than just individual words. This may improve search accuracy, especially when dealing with multiple terms.
- Using NER algorithms: allow the system to identify and categorize entities within a text, such as company's name, locations, dates, phone numbers (and relative alignment for numbers with and without country code) and other important terms. By assigning a higher weight to these entities, the retrieval system can give higher or lower rank results based on the significance of these keywords.
- Test reranking more extensively, maybe reducing the documents on which it is applied, or just give it more time.
- Examine the decline in performance following stopwords removal by exploring potential causes and verifying the code for errors, given that this is an unusual occurrence, although it has been documented in the literature [9].
- Translating all documents into French, if they are not already: translating all documents into French ensures that all documents are processed uniformly, allowing the system to apply consistent tokenization and linguistic rules. This can lead to better matching of queries with documents, but it can also introduce translation errors or nuances that could affect the retrieval quality. However, as described in the relative section, this approach would require significant computational resources and time, especially for large datasets.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-4o and Writefull in order to: Improve writing style and Grammar and spelling check. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Cancellieri, A. El-Ebshihy, T. Fink, P. Galuščáková, G. Gonzalez-Saez, L. Goeuriot, D. Iommi, J. Keller, P. Knoth, P. Mulhem, F. Piroi, D. Pride, P. Schaer, Overview of the CLEF 2025 LongEval Lab on Longitudinal Evaluation of Model Performance, in: J. Carrillo-de Albornoz, J. Gonzalo, L. Plaza, A. García Seco de Herrera, J. Mothe, F. Piroi, P. Rosso, D. Spina, G. Faggioli, N. Ferro (Eds.),

Experimental IR Meets Multilinguality, Multimodality, and Interaction. Proceedings of the Sixteenth International Conference of the CLEF Association (CLEF 2025), 2025.

[2] I. Atabek, H. Chen, J. Moncada Ramírez, N. Santini, G. Zago, Seupd@clef: Team JIHUMING on enhancing search engine performance with character n-grams, query expansion, and named entity recognition, in: Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2023), Thessaloniki, Greece, September 18th to 21st, 2023, volume 3497 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 2204–2221. URL: https://ceur-ws.org/Vol-3497/paper-185.pdf.

[3] S. Bortolin, G. Ceccon, G. Czaczkes, A. Pastore, P. Renna, G. Zerbo, Seupd@clef: Team NEON. a memoryless approach to longitudinal evaluation, in: Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2023), Thessaloniki, Greece, September 18th to 21st, 2023, volume 3497 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 2281–2305. URL: https://ceur-ws.org/Vol-3497/paper-189.pdf.

[4] S. Fincato, E. D'Alberton, Y. Qiu, L. Vaidas, L. Pallante, A. Jassal, Seupd@clef: Team QEVALS on information retrieval adapted to the temporal evolution of web documents, in: Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2023), Thessaloniki, Greece, September 18th to 21st, 2023, volume 3497 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 2416–2431. URL: https://ceur-ws.org/Vol-3497/paper-194.pdf.

[5] F. Galli, M. Rigobello, M. Schibuola, R. Zuech, N. Ferro, Seupd@clef: Team IRIS on temporal evolution of query expansion and rank fusion techniques applied to cross-encoder re-rankers, in: Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2024), Grenoble, France, 9-12 September, 2024, volume 3740 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 2356–2383. URL: https://ceur-ws.org/Vol-3740/paper-218.pdf.

[6] L. Cazzador, F. L. De Faveri, F. Franceschini, L. Pamio, S. Piron, N. Ferro, Seupd@clef: Team MOUSE on enhancing search engines effectiveness with large language models., in: Working Notes of the Conference and Labs of the Evaluation Forum (CLEF 2024), Grenoble, France, 9-12 September, 2024, volume 3740 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 2336–2355. URL: https://ceur-ws.org/Vol-3740/paper-217.pdf.

[7] J. Savoy, Light Stemming Approaches for the French, Portuguese, German and Hungarian Languages, in: H. M. Haddad, K. M. Liebrock, R. Chbeir, M. J. Palakal, S. Ossowski, K. Yetongnoon, R. L. Wainwright, C. Nicolle (Eds.), Proc. 21st ACM Symposium on Applied Computing (SAC 2006), ACM Press, New York, USA, 2006, pp. 1031–1035.

[8] G. Salton, M. E. Lesk, The SMART automatic document retrieval systems — an illustration, Commun. ACM 8 (1965) 391–398. URL: https://doi.org/10.1145/364955.364990. doi:10.1145/364955.364990.

[9] J. Savoy, A Stemming Procedure and Stopword List for General French Corpora, Journal of the American Society for Information Science (JASIS) 50 (1999) 944–952.

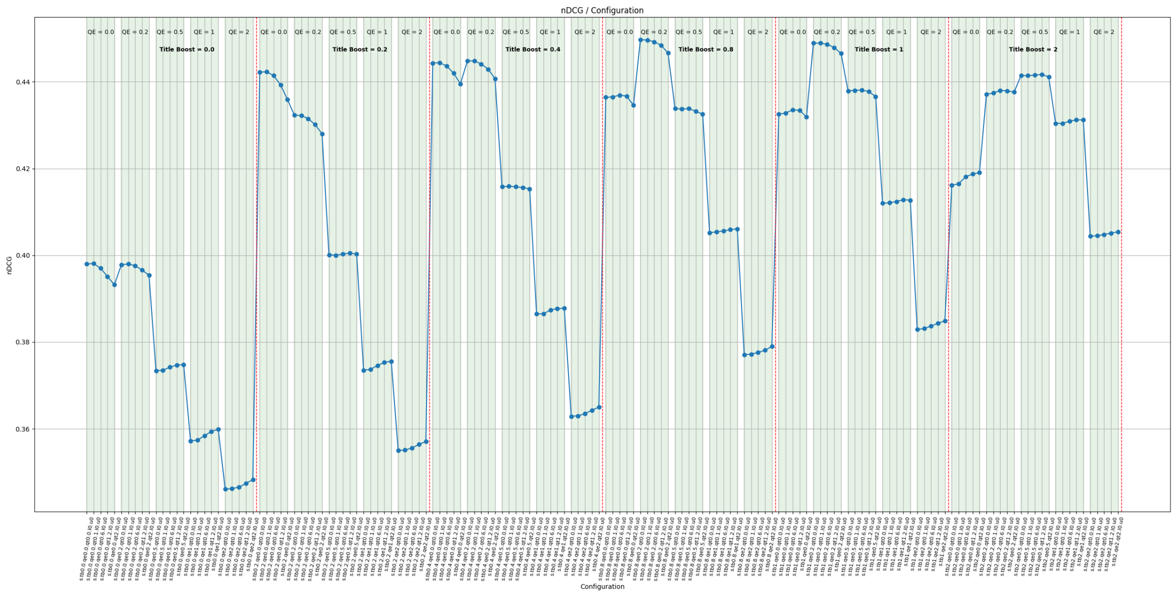


Figure 8: Grid search on TitleBoost, Query Expansion, and Query Translation

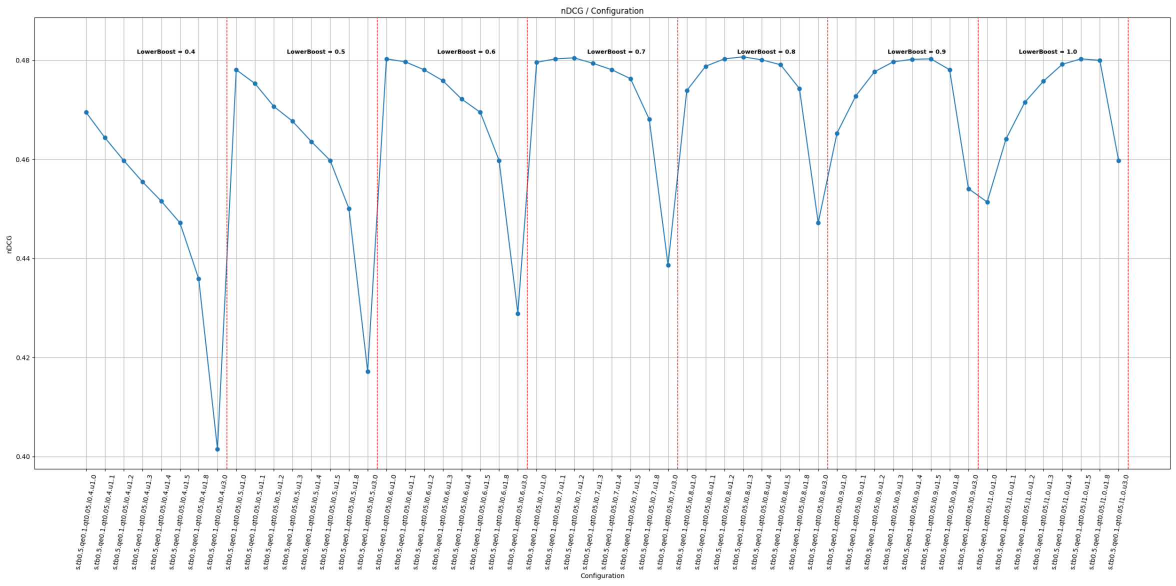


Figure 9: Results of the grid search for URL-Boost