SEUPD@CLEF: Team Searchill

Notebook for the LongEval Lab at CLEF 2025

Alessandro Di Frenna[1], Luca Pellegrini[1] and Nicola Ferro[1]

[1]*University of Padua, Italy*

Abstract

This report describes the search engine developed by the team "Searchill" for the LongEval-Retrieval Task 1 at the CLEF 2025 conference. The main objective of the project was to develop an effective yet efficient search engine. To achieve this goal, our team has extensively experimented with various techniques and approaches. To assess system performance, we used the CLEF corpus and human judgements. Our top-performing system, which combines traditional methods with AI, has encouraging results and might be the basis for further advancements in the field.

Keywords

CLEF 2025, LongEval, Information Retrieval, Search Engine

1. Introduction

This research endeavors to establish a robust retrieval system capable of adapting to the inherently dynamic nature of the internet. Our methodology leverages the comprehensive Longeval Websearch collection [?], an expansive dataset encompassing online pages, thematic topics, and user interaction patterns. This initiative has significantly advanced our comprehension of information retrieval paradigms within the context of contemporary search engines. Notably, we prioritized refining both query and document processing methodologies to optimize ranking efficacy and ensure the delivery of highly pertinent information to users.

The structure of this paper is organized as follows: Section 2 delineates our proposed approach; Section 3 details the experimental configuration; Section 4 presents and discusses our principal findings; finally, Section 5 offers concluding remarks and outlines directions for future research.

1.1. Related Work

Guided by the instructional sessions from the SearchEngines course at the University of Padua, we proceeded with the systematic development of the core components—namely, the indexer, analyzer, and searcher—adhering to the architectural framework introduced and thoroughly examined during the class.

2. Methodology

In this section, we describe the methodology adopted to develop our IR system for the task.

2.1. Parser

The parser is responsible for reading and extracting structured data from complex document formats, enabling effective indexing and retrieval. Our document parsing framework consists of several components designed to handle different document structures efficiently:

CLEF 2025: Conference and Labs of the Evaluation Forum, September 9–12, 2025, Madrid, Spain

alessandro.difrenna@studenti.unipd.it (A. D. Frenna); luca.pellegrini@studenti.unipd.it (L. Pellegrini); nicola.ferro@unipd.it (N. Ferro)
https://www.dei.unipd.it/~ferro/ (N. Ferro)
0000-0001-9219-6239 (N. Ferro)

- **DocumentParser:** This abstract class serves as the core of our parsing framework. It provides a template for iterating over documents. The class ensures that documents are read correctly.
- **ParsedDocument:** This class encapsulates the structure of a parsed document. It defines essential fields such as:
  - `ID` - A unique identifier for each document.
  - `Body` - The main content of the document.
  - `Start` - Initial lines of the document, often used for quick reference.
  - `Highlights` - Notable words or phrases extracted from tags or special formatting, enhancing keyword extraction.
- **TrecParser:** This specialized class extends `DocumentParser` to handle TREC-formatted documents. Key functions include:
  - Regex-based extraction of document fields such as `<DOCNO>` and `<DOCID>`.
  - Management of document content using a multistage buffering approach to efficiently handle large text bodies.
  - Extraction of emphasized content through patterns such as `hashtags` and `strong` tags, improving the semantic richness of stored highlights.
  - Support for emoji and URL filtering, ensuring text data is clean and normalized before analysis.
- **Error Handling:** Robust error management is implemented to gracefully handle missing or malformed data, maintaining the stability and reliability of the parser.

This structured parsing methodology allows for the seamless transformation of raw text into well-defined data objects, paving the way for accurate indexing and retrieval operations.

2.2. Analyzer

The `FrenchAnalyzer` is designed to process French texts, integrating multiple components:

In the construction of a Lucene-based text analysis pipeline, the sequence of token filters applied to a token stream plays a critical role in shaping the effectiveness of indexing and retrieval. This document outlines the rationale behind the sequential ordering of token filters implemented in the `createComponents` method of the analyzer. Each filter serves a distinct purpose, and its placement in the pipeline is determined by its functional dependencies and intended effects on the token stream.

Pipeline Overview:

The method under consideration constructs the analysis pipeline as follows:

1. A tokenizer is selected based on the `tokenizerType`, producing the initial token stream.
2. A sequence of token filters is applied in a specific order:
   a) `LowerCaseFilter`
   b) Optional: `RepeatedLetterFilter`
   c) Optional: `AbbreviationExpansionFilter`
   d) `ICUFoldingFilter`
   e) `NBSPFilter`
   f) `ElisionFilter`
   g) `RemoveDuplicatesTokenFilter`
   h) Optional: `CompoundPOSTokenFilter`
   i) Optional: `ShingleFilter`
   j) Optional: `LengthFilter`
   k) `Stopword Removal`
   l) Optional: `PositionFilter`

2.3. LowerCaseFilter

The `LowerCaseFilter` is applied immediately after tokenization to normalize case, ensuring that all subsequent filters operate on a consistent lowercase representation. This step facilitates reliable term matching and avoids case-sensitive inconsistencies in abbreviation expansion or folding.

2.4. RepeatedLetterFilter (Optional)

When enabled, the `RepeatedLetterFilter` addresses exaggerated letter repetitions that might arise from informal language or user typing quirks. Placing this filter early ensures that such noise does not affect subsequent expansion, folding, or duplicate removal.

2.5. AbbreviationExpansionFilter (Optional)

This filter expands known abbreviations using a predefined abbreviation map.

In information retrieval and natural language processing, abbreviations are frequent sources of ambiguity. Expanding abbreviations to their full forms can enhance both search accuracy and linguistic analysis. This document describes the implementation logic of an abbreviation expansion filter in a Lucene-based text processing pipeline, detailing its steps and rationale. For each token, it performs the following:

- Checks if the lowercase form of the token exists in a provided abbreviation map.
- If found, replaces the token with its expanded form.
- Otherwise, leaves the token unchanged.

Case Insensitivity By converting tokens to lowercase before lookup, the filter accommodates abbreviations written in different cases (e.g., “NLP”, “nlp”, “Nlp”), ensuring consistent expansion regardless of text casing.

Pipeline Integration Since the filter only modifies tokens when a match is found, it preserves the behavior of downstream filters, ensuring modular and reusable analysis components.

2.6. ICUFoldingFilter

The ICU folding filter is applied to harmonize Unicode representations, handling diacritics, special characters, and other script variations. Placing this filter at this stage ensures that downstream filters operate on a canonicalized representation.

2.7. NBSPFilter

The `NBSPFilter` normalizes non-breaking spaces and related whitespace anomalies that may affect token boundaries. Applying it after Unicode folding ensures consistent spacing.

2.8. ElisionFilter

This filter removes common French elisions (e.g., “l’avion” $\rightarrow$ “avion”). It is applied after normalization steps to ensure that contractions and elided forms are treated as their base terms.

2.9. RemoveDuplicatesTokenFilter

Duplicate tokens are removed at this stage to avoid term inflation that could bias scoring or relevance models.

2.10. CompoundPOSTokenFilter (Optional)

In natural language processing (NLP), grouping or combining adjacent tokens based on their part-of-speech (POS) tags is a common technique to identify and process meaningful multi-word expressions (MWEs), compound nouns, and other linguistic constructs. This document presents a detailed explanation of the logic implemented in the provided token filter, which applies POS-based token grouping within a Lucene analysis pipeline. The filter operates by:

- Buffering all tokens from the input token stream.
- Assigning POS tags to the buffered tokens using a pre-trained POS tagger.
- Combining tokens based on specific POS tag patterns.
- Reconstructing a new token buffer containing the grouped tokens.
- Returning the next token on demand.

Initially, all tokens from the input stream are buffered. This ensures that the POS tagger can analyze the entire sentence or sequence, rather than processing tokens in isolation. This global view is essential for capturing meaningful patterns such as compound nouns or adjective-noun sequences.

2.10.1. POS Tag Patterns for Grouping

The core logic groups tokens based on a set of pre-defined POS tag patterns:

- **Three-token patterns**:
  - NC-NC-NC: Three consecutive common nouns (e.g., *"carte bancaire dette"*).
  - NC-P-NC: Noun + Preposition + Noun (e.g., *"président de société "*).
- **Two-token patterns**:
  - NC-NC, N-N: Consecutive nouns (e.g., *"ingénieur logiciel"*).
  - V-NC: Verb followed by a noun (e.g., *"prendre pause"*).
  - ADJ-ADJ: Consecutive adjectives (e.g., *"grand rouge "*).
  - NC-ADJ: Noun followed by an adjective (e.g., *"pomme verte"*).

When a match is detected:

1. The relevant tokens are concatenated using a hyphen (e.g., "ingénieur logiciel").
2. The token type is set to NN, indicating a compound noun.
3. The grouped token is added to the new buffer.

If no pattern matches, the token is added as is, with its corresponding POS tag as its type.

The processing order ensures that:

1. POS tagging occurs after all tokens are collected, ensuring correct context-aware tagging.
2. Grouping is performed from the start of the sequence, respecting the natural left-to-right order of language.
3. Longer groupings (three-token patterns) are prioritized over shorter ones to avoid prematurely splitting sequences that could form meaningful MWEs.
4. The type attribute assignment (NN) standardizes the representation of multi-word expressions in downstream analysis.

This token filter exemplifies an effective method for combining tokens based on syntactic criteria using POS tags. The design ensures that the resulting token stream captures meaningful multi-word expressions, enhancing the quality of text analysis and retrieval within a Lucene-based system.

2.11. ShingleFilter (Optional)

This filter generates n-grams (shingles) from the token stream. It is placed after POS-based filtering to capture meaningful multi-word units. Applying it before length or position filters ensures that n-grams are subject to subsequent size and position constraints.

2.12. LengthFilter (Optional)

This filter restricts tokens by minimum and maximum length. Placing it after n-gram generation ensures that composite tokens are also evaluated for length.

2.13. Stopword Removal:

The `StopFilter`, loaded with a list of irrelevant tokens, removes common stopwords that do not contribute to the semantic meaning of texts and queries

2.14. PositionFilter (Optional)

Finally, the `PositionFilter` adjusts token position increments for downstream analysis or ranking. As the last step, it operates on the fully processed token stream.

The sequence of token filters in the `createComponents` method is carefully designed to ensure that each filter operates on a consistently normalized and preprocessed token stream. This design maximizes the effectiveness of each filter while preserving the linguistic and statistical integrity of the analysis pipeline.

2.15. Stemming:

customised to handle French morphology:

- `SnowballFilter`(snow) - Employs the Snowball stemming algorithm, recognized for its effectiveness across languages.
- `FrenchLightStemFilter` Applies a light stem, suitable for texts requiring minimal conversion.
- `FrenchMinimalStemFilter` Executes minimal stemming, avoiding over-reduction

  French Minimal Stemmer The *French Minimal Stemmer* is even simpler: it removes only a few suffixes, aiming to minimally transform words. It is ideal in contexts where retaining the original word root is crucial, but is less effective at reducing morphological variants.

2.16. Tokenization:

We offer a selection of tokenizer modules designed to accommodate diverse text processing requirements:

- `WhitespaceTokenizer` – This tokenizer segments text based on whitespace delimiters, proving optimal for inputs with inherent structural formatting.
- `LetterTokenizer` – It separates tokens at any non-alphabetic character, making it particularly suited for the analysis of purely alphabetic textual content.
- `StandardTokenizer` – This component utilizes sophisticated parsing rules to manage intricate textual constructs, including the proper handling of punctuation and special characters.
- `OpenNLPTokenizer` – This option employs statistical models provided by OpenNLP for tokenization, a method highly advantageous for processing complex linguistic structures.

2.17. Indexer

- **BodyField**: represents the main content field of a document:
  - **token**: the body text is split into tokens.
  - **frequency**: the frequency of tokens is recorded.
  - **position**: the positions of tokens are stored to support phrase queries.
  - **body store**: the entire document body is stored to enable re-ranking.
- **StartField**:
  - **token**: the start field is tokenized.

  - **frequency**: the frequency of tokens in the start field is stored.
  - **body store**: the initial portion of the document is saved.
- **HighlightedField**:
  - **token**: the highlighted sections are tokenized.
  - **frequency**: the frequency of tokens in the highlights is recorded.
  - **body store**: the highlights of the document are stored.
- **DirectoryIndexer**: This class is responsible for indexing documents within a specified directory. It supports several configuration parameters, such as the directory path, the parser type used to process documents, the analyzer for text processing, the similarity metric applied during indexing, the expected number of documents, and the location where the final index is saved.

Collection	Docs Size (GB)	Body Terms	Number of Documents	Index Size (GB)
French 2022-06	9,3	3,423,470	1,590,024	7.5
French 2023-01	12.8	38,313,777	2,537,554	12.4

Table 1
Indexing output details.

2.17.1. StopList Choice

Subsequent to the indexing of all documents within the training set, we utilized LUKE, a utility provided by Lucene. This tool enabled us to thoroughly inspect our indexed documents, gaining insights into their indexing structure and facilitating the identification of the most frequently occurring tokens within our collection.

In an effort to optimize performance, we proceeded to construct diverse stop lists. These lists incorporated various combinations, specifically differentiating between numerical terms, English linguistic terms, or a blend of both. We then rigorously analyzed the computational outcomes derived from each of these distinct configurations.

We chose the best one, namely "stopList40FR-10EN", composed by the first 40 French terms and 10 English terms. We decided to use it for all the runs.

Stemmer tuning:

Run name 2022-06	nDCG	MAP
stem-FrenchLight	0.1398	0.0703
stem-SnowBall	0.1499	0.0767
stem-FrenchMinimal	0.1426	0.0722

Table 2
Stemmer tuning 2022-06

Run name 2023-01	nDCG	MAP
stem-FrenchLight	0.2207	0.1345
stem-SnowBall	0.2315	0.1422
stem-FrenchMinimal	0.2250	0.1370

Table 3
Stemmer tuning 2023-01

Tokenizer tuning:

Run name 2022-06	nDCG	MAP
tokenizer-WhiteSpace	0.1541	0.0796
tokenizer-Letter	0.1723	0.0923
tokenizer-Standard	0.1710	0.0919

Table 4
Tokenizer tuning 2022-06

Run name 2023-01	nDCG	MAP
tokenizer-WhiteSpace	0.2420	0.1512
tokenizer-Letter	0.2675	0.1710
tokenizer-Standard	0.2656	0.1695

Table 5
Tokenizer tuning 2023-01

2.18. Searcher

Within the searcher component, we employed a Boolean query strategy to support query expansion and improve the overall performance of the information retrieval system. The initial query terms were marked as mandatory for document retrieval, while the expanded terms were added as optional elements, allowing for greater flexibility in matching relevant documents. Additionally, we applied a boosting mechanism to assign higher weight to the original query, ensuring it remained central to the retrieval process. The following section provides a detailed explanation of the implemented query expansion methodology.

2.18.1. Start

It is often observed that the primary subject of a document is introduced at the beginning of the text. Based on this assumption, we designed the indexer to store the first five lines of each document. This design choice allowed the search engine to perform query operations specifically on this initial segment.

2.18.2. Word N-grams

As the first step in our query expansion approach, we used word N-grams—sequences of consecutive terms extracted from the original query—to enrich the query representation. This process involves tokenizing the initial query into individual words and then recombining them into adjacent word pairs. For example, the query running shoes women is tokenized into running, shoes, and women, from which we generate the bigrams running shoes and shoes women.

2.18.3. Fuzzy Search

To enhance the robustness of our search system, we integrated a fuzzy search mechanism that supports approximate matches between query terms and document content. This technique is especially effective in addressing minor input errors, such as typos, letter transpositions, or common abbreviations. By setting a similarity threshold, the system ensures that only plausible variants of the query terms are considered, maintaining an acceptable level of accuracy. For example, a query like Adibas chaussures could successfully retrieve documents containing the correctly spelled brand name Adidas.

2.18.4. Dictionary

During the analysis of user input, we found that some queries appeared in unconventional formats, such as URLs, dot-separated terms (e.g., term1.term2.term3), or concatenated words without spaces (e.g., term1term2term3), which complicated processing by the search engine. To handle issues caused by punctuation, we implemented a regular-expression-based function to systematically remove these characters from the input.

For concatenated terms, we employed a dictionary derived from the provided synonyms file. This dictionary enables the system to identify and extract meaningful sub-terms embedded within single-word queries.

2.18.5. Synonyms

Implementing synonyms in retrieval presents challenges, including semantic ambiguity and noise due to polysemy and irrelevant matches. While synonyms theoretically broaden the search scope, actual improvements depend on the relevance of the context. To manage computational overhead, synonyms were handled at query time using a query expansion model. This approach aims to improve retrieval accuracy while keeping the system efficient.

Synonyms Tuning: Since the synonyms approach randomly selects three synonyms from the collection, we decided not to apply the same experimental testing as we did with the Fuzzy and Phrase modules, due to the lack of repeatability introduced by the random selection process.

2.18.6. Phrase Query

To further improve the precision of our retrieval system, we implemented a Phrase Query Searcher that identifies documents containing an exact sequence of terms in the order they appear in the query. Phrase queries require that the specified terms occur in contiguous positions within the document.

2.18.7. Word2Vec

To support word similarity functionality via a web-accessible interface, we developed a lightweight HTTP server using the Flask framework, chosen for its simplicity and ease of deployment in small-scale applications. At the core of the system is the KeyedVectors interface from the Gensim library, which allows efficient loading and querying of pre-trained word embedding models.

For our implementation, we used 300-dimensional French fastText word vectors (cc.fr.300.vec), trained on the large-scale Common Crawl corpus. These vectors are especially suitable for capturing semantic relationships between French words, thanks to their subword-based architecture that improves performance on rare or morphologically rich terms.

The main purpose of this service is to support query expansion within our information retrieval pipeline. When a user submits a search query, the system sends the main term to the /similar endpoint.

2.18.8. LLM

LLM-based query expansion is a technique used to improve the effectiveness of a search engine by automatically expanding the original user query. This approach is particularly useful when user input is too generic, vague, or ambiguous to return truly relevant results.

In our implementation, the query is sent to a Large Language Model (LLM), specifically ChatGPT-3.5-turbo, which interprets the meaning of the sentence and generates three to four related terms.

This number of terms was chosen strategically: too many would introduce information noise, potentially reducing precision, while too few would compromise the effectiveness of the technique.

The choice of using ChatGPT-3.5-turbo was driven by a trade-off between performance and cost.

2.18.9. Query Construction

The core search process employs a dynamic Boolean query builder with weighted clauses:

- **Base Query**: Mandatory clause parsed from the original query string
- **Field Boosting**:
  - Start field matches boosted by 1.5×
  - Highlight field matches boosted by 1.2×
- **Query Expansion**: Optional features:
  - Dictionary-based synonyms (0.1× weight)
  - Fuzzy search with edit distance=2 (0.2×-0.8× weights), which allows for typos
  - Phrase queries with slop=5 window

2.18.10. Query Expansion Techniques

We implemented six complementary expansion strategies:

- **Word N-grams Generation**: Generates bigrams from query terms to capture word associations.
- **Synonym Expansion**: Uses dictionary-based synonyms with boosted weights:
  - Start field matches boosted by 1.5×
  - Highlight field matches boosted by 1.2×
- **Neural Expansion**:

  - LLM-based term generation via GPT-3.5/DeepSeek APIs
  - Requests 5-10 related terms per query
  - Applies 0.4× weight to generated terms
- **Pseudo-Relevance Feedback (PRF)**: Uses top-ranked documents to expand queries with relevant terms.

2.18.11. Re-ranking Pipeline

Final results undergo neural re-ranking:

1. Normalize initial scores to [0,1] range
2. Retrieve top 50 document bodies
3. Invoke Cohere multilingual-v3 re-ranker or local reranker model
4. Combine scores using linear interpolation:

$$\mathit{finalScore} = 0.4 \cdot \mathit{BM25} + 0.6 \cdot \mathit{Reranker}$$

2.18.12. Heuristic Reranker

In the context of information retrieval systems, the `HeuristicReranker` module is designed to adjust the relevance scores of retrieved documents based on several heuristic penalties and bonuses. This aims to improve the ranking quality by applying domain-specific adjustments that consider document length, vocabulary richness, repetition, and potential spam.

The module extracts terms and their frequencies from Lucene term vectors:

- `getDocumentTerms()` collects term frequencies for each document using `IndexReader` and stores them in a `Map`.

Heuristic Penalties:

Several penalties are applied to rerank documents:

- **Length Penalty**: Penalizes documents that are too short or excessively long, to prioritize well-structured documents.
- **Vocabulary Penalty**: Assesses the lexical richness of a document (ratio of unique terms to total terms). Documents with low richness are penalized, while highly diverse documents receive a score bonus.
- **Spam Term Frequency Penalty**: Penalizes documents that exhibit keyword stuffing, i.e., where query terms are overrepresented relative to the total document length.
- **Repetition Penalty**: Penalizes documents with high term repetition, which might indicate low-quality or spammy content.

Query Heuristics:

The module also implements simple query heuristics:

- For short queries (two terms or fewer), documents that are longer and lexically rich are boosted.
- For queries with diverse terms, documents with high lexical richness are also boosted.

3. Experimental Setup

The experiments were conducted on a Asus Vivobook X515 with the following hardware specifications:

- **CPU:** AMD Ryzen 5500U, 6-core
- **GPU:** Radeon Graphics

- **RAM:** 12 GB LPDDR5, 3200 MHz

The datasets utilized in this study are those made available by the LongEval CLEF 2025 Lab, accessible at: https://clef-longeval.github.io/data/

The full implementation of the project is available at: https://bitbucket.org/upd-dei-stud-prj/seupd2425-searchill/src/master/ — Bitbucket

To use the query correction feature, you can either use the LanguageTool API available at https://api.languagetoolplus.com/v2 or run a local instance available at https://dev.languagetool.org/http-server.

The reranker is used either via a local Python HTTP server or through the Cohere web service (https://cohere.com/), which was employed for comparison purposes but is limited in the number of requests.

4. Train Results

In this section we present the results obtained by our systems. In particular, Section 4 summarizes the system's performance with different configurations on the training dataset.

We evaluated different configurations of our search engine by integrating the components described in Section 2 and tuning their parameters for optimal performance. Table **??** reports the MAP and nDCG scores for each method, based on test data and relevance judgments of the 2023/01 dataset.

Table 6
Evaluation results on MAP and NDCG for different methods

Method	MAP	NDCG
Tokenizers		
Whitespace	0.1855	0.2847
Letter	**0.2002**	**0.3061**
Standard	0.2002	0.3059
NLP	0.1892	0.2906
Stemmers		
FrenchLight	0.1722	0.2745
FrenchMinimal	0.1798	0.2832
Snowball	0.1848	0.2903
Analyzer methods		
Expansion	0.2000	0.3058
Position	0.2002	0.3061
Double	0.2003	0.3063
PositionOpnNLP	**0.2011**	**0.3069**
Other Methods		
Dictionary	0.2014	0.3072
Synonyms	0.1979	0.3034
Fuzzy	0.2135	0.3230
Phrase	0.2243	0.3270
PRF	0.1768	0.2905
Word2vec	0.1689	0.2814
N-grams	0.2002	0.3061
Heuristic	0.1361	0.2237
Start	0.2011	0.3069
Highlights	0.2002	0.3062
Phrase + Fuzzy + Start	0.2359	0.3422
Phrase + Fuzzy + Start + Highlights	0.2359	0.3423
Phrase + Fuzzy + Start + N-grams	0.2359	0.3422
Phrase + Fuzzy + Start + dictionary	0.2355	0.3420
Phrase + Fuzzy + Start + Highlights + Doubles	0.2359	0.3423
Phrase + Fuzzy + Start + Highlights + Reranker	0.2585	0.3600
Phrase + Fuzzy + Start + Highlights + Doubles + Reranker	**0.2585**	**0.3601**

Bold numbers indicate the best-performing configuration. All methods in each section use the best parameter values from the preceding sections.

Tokenizers:

Among the tokenizers, the Letter Tokenizer achieved the best results, slightly outperforming the Standard Tokenizer. In contrast, the Whitespace Tokenizer yielded poorer retrieval performance, while the NLP tokenizer was slower and less effective due to its computational overhead.

Stemmers:

All stemmers negatively impacted retrieval performance and were therefore excluded from subsequent configurations.

Analyzer Methods:

Table 7
nDCG and MAP values - Baseline: Phrase + Fuzzy + Start + Highlights

Date	Baseline	Baseline + Reranker_50	Baseline + Correction + Reranker_50
2023/3	0.2394 - 0.3449	0.2706 - 0.3703	0.2707 - 0.3704
2023/4	0.2566 - 0.3682	0.2804 - 0.3865	0.2800 - 0.3861
2023/5	0.2515 - 0.3603	0.2802 - 0.3830	0.2796 - 0.3823
2023/6	0.2685 - 0.3756	0.2973 - 0.3985	0.2970 - 0.3982
2023/7	0.2475 - 0.3539	0.2715 - 0.3736	0.2706 - 0.3725
2023/8	0.2208 - 0.3095	0.2375 - 0.3209	0.2355 - 0.3207

We extended our analysis using the Letter Tokenizer and stopword filtering as a baseline, applying expansion, position-based weighting, and OpenNLP-based position tagging. Among these, only OpenNLP position tagging provided a performance improvement.

Other Methods:
To enable a systematic and manageable evaluation, each method was primarily tested in isolation under the simplifying assumption that components do not interact. This approach allowed us to isolate and measure the individual contribution of each feature. Features that did not yield improvements were excluded from subsequent configurations.
Neutral or beneficial features were tested in combination to assess potential performance gains. In the case of the dictionary, although it showed improvement in isolation, its integration with other configurations led to a decrease in performance.
The combination of Phrase, Fuzzy, Start, Highlights, and Reranker yielded the best results, and its features were adopted in the following evaluation stages.

5. Test Results

In this Section we will show and analyze the results obtained by our system of the runs on the test collections.

Table 8
ANOVA2 on heldout collection

	nDCG 2023/03					AP 2023/03				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	2.2037	2	1.1018	99.2415	1.11e-16	3.3111	2	1.6556	104.3786	1.11e-16
Topics	1516.7022	5091	0.2979	26.8332	1.11e-16	1656.7083	5091	0.3254	20.5169	1.11e-16
Error	113.0466	10182	0.0111	-	-	161.4973	10182	0.0159	-	-
Total	1631.9525	15275	-	-	-	1821.5167	15275	-	-	-

	nDCG 2023/04					AP 2023/04				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	2.8477	2	1.4238	142.4028	1.11e-16	4.8140	2	2.4070	167.9370	1.11e-16
Topics	3833.4804	13019	0.2945	29.4491	1.11e-16	4280.9664	13019	0.3288	22.9423	1.11e-16
Error	260.3461	26038	0.0100	-	-	373.1943	26038	0.0143	-	-
Total	4096.6742	39059	-	-	-	4658.9747	39059	-	-	-

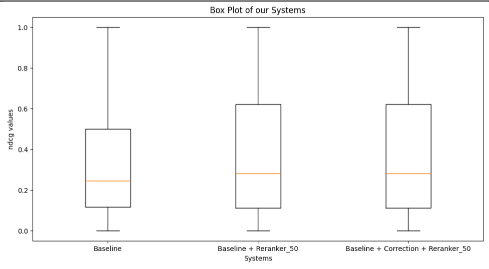
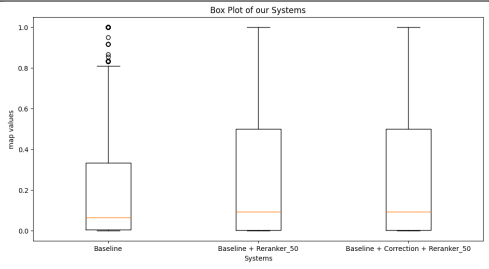
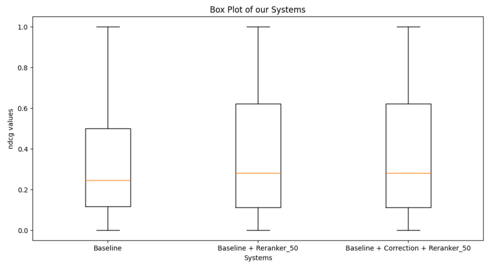
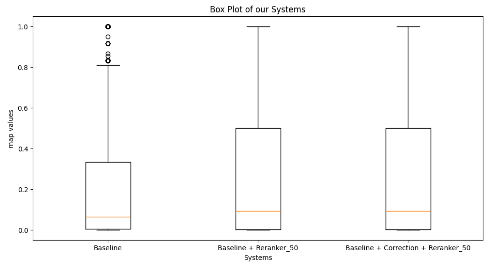
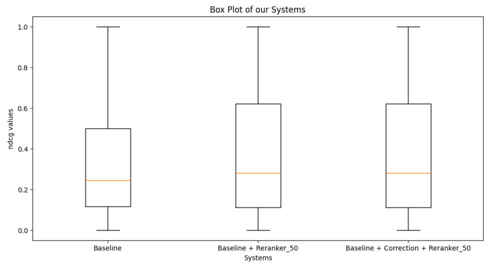
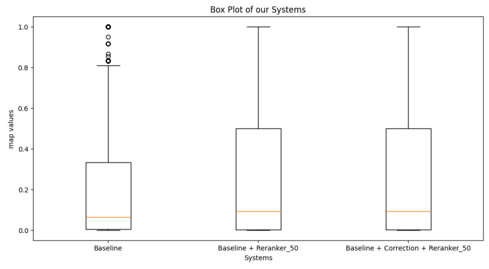
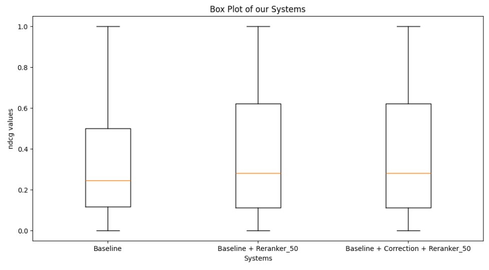
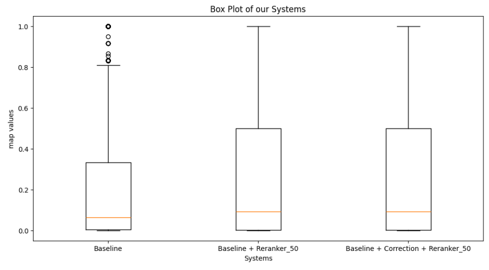
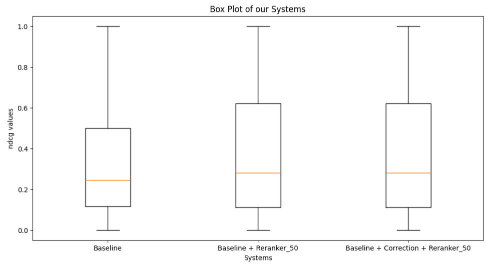
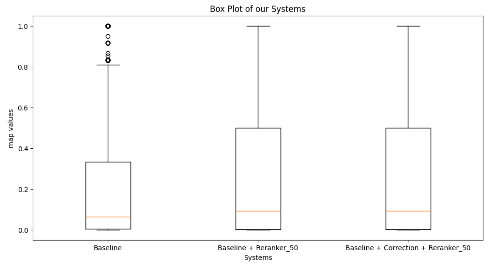
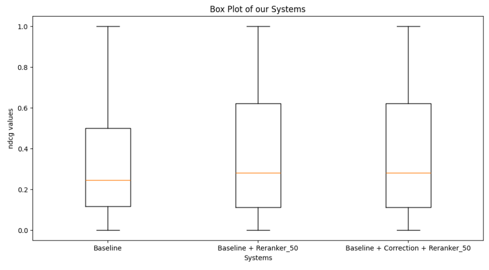
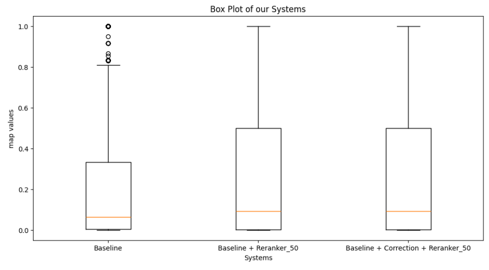
	nDCG 2023/05					AP 2023/05				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	3.5115	2	1.7557	180.8835	1.11e-16	5.6599	2	2.8300	201.4644	1.11e-16
Topics	3137.5278	10540	0.2977	30.6681	1.11e-16	3475.6841	10540	0.3298	23.4756	1.11e-16
Error	204.6118	21080	0.0097	-	-	296.1097	21080	0.0140	-	-
Total	3345.6511	31622	-	-	-	3777.4538	31622	-	-	-

	nDCG 2023/06					AP 2023/06				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	2.4794	2	1.2397	100.8968	1.11e-16	3.9272	2	1.9636	110.6254	1.11e-16
Topics	2233.7190	7203	0.3101	25.2389	1.11e-16	2471.6517	7203	0.3431	19.3319	1.11e-16
Error	177.0063	14406	0.0123	-	-	255.7065	14406	0.0178	-	-
Total	2413.2047	21611	-	-	-	2731.2854	21611	-	-	-

	nDCG 2023/07					AP 2023/07				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	2.1803	2	1.0902	110.1607	1.11e-16	3.3034	2	1.6517	115.5261	1.11e-16
Topics	2652.5206	8946	0.2965	29.9616	1.11e-16	2857.5674	8946	0.3194	22.3420	1.11e-16
Error	177.0613	17892	0.0099	-	-	255.8024	17892	0.0143	-	-
Total	2831.7623	26840	-	-	-	3116.6731	26840	-	-	-

	nDCG 2023/08					AP 2023/08				
Source	**SS**	**df**	**MS**	**F**	**Prob>F**	**SS**	**df**	**MS**	**F**	**Prob>F**
Systems	0.9767	2	0.4884	60.6778	1.11e-16	1.6796	2	0.8398	74.3744	1.11e-16
Topics	3496.3967	11551	0.3027	37.6084	1.11e-16	3418.7098	11551	0.2960	26.2108	1.11e-16
Error	185.9371	23102	0.0080	-	-	260.8629	23102	0.0113	-	-
Total	3683.3105	34655	-	-	-	3681.2524	34655	-	-	-

5.0.1. explanations

Data	nDCG	AP
2023/03		
2023/04		
2023/05		
2023/06		
2023/07		
2023/08		

The baseline system (Phrase + Fuzzy + Start + Highlights) shows consistent nDCG values ranging approximately from 0.22 to 0.27 and MAP values from 0.31 to 0.38 across the months March to August 2023.

Adding the `Reranker_50` improves both nDCG and MAP scores noticeably in all months compared to the baseline.

The combination of `Correction + Reranker_50` either matches or slightly improves over the `Reranker_50` alone.

Overall, reranking methods enhance retrieval quality significantly over the baseline.

For every month (March to August 2023), the ANOVA tests on heldout collections show:

- Highly significant effects of the system ($p < 0.0001$) on both nDCG and AP, indicating that different systems (Baseline, Baseline + Reranker, Baseline + Correction + Reranker) perform differently with high confidence.
- Significant variance attributed to topics, confirming topic-level differences.
- Relatively low error variance compared to the effects, supporting reliability of the results.

August showed lower performance compared to the other months, with no clear explanation.

6. Conclusions and Future Work

The optimal training configuration integrates the following components: Phrase, Fuzzy matching, Start anchoring, Highlights, and a Reranker module. Significant performance improvements are primarily driven by the inclusion of Fuzzy matching, Phrase matching, and the Reranker component.

Our system does not significantly benefit from basic information retrieval features such as query expansion, word2vec embeddings, n-grams, synonyms, or dictionaries, primarily due to suboptimal performance and the difficulty of identifying suitable parameters for our dataset. As a result, the overall performance of these techniques remained mediocre, though they still allowed for a baseline level of retrieval capability.
Heuristic-based methods demonstrate poor effectiveness, reflected in the lowest MAP scores, likely due to the lack of an optimal parameter combination or because the overlap of multiple filters causes their effects to cancel each other out. The Part-of-Speech (POS) model used, given the very limited set of grammar rules combinations, is too weak to effectively identify all compound words. To address this, a Named Entity Recognition (NER) model should have been incorporated to perform a similar role; however, Apache OpenNLP no longer provides a suitable NER model.

Due to limited computational resources, we were unable to experiment with more advanced reranking methods or implement query expansion techniques based on large language models, which we believe could substantially improve retrieval performance. Furthermore, we could not include a feature designed to extract a small number of representative keywords from each document to support more effective search, again due to resource constraints.

7. Group Members Contribution

Alessandro Di Frenna Tested and evaluated various tokenizers. Implemented AbbreviationExpansionFilter and CompoundPOSTokenFilter using Apache OpenNLP library. Implementing HeuristicReranker

Luca Pellegrini Responsible for the development of the parser, start and highlight features, and the construction of the stoplist. Performed tuning and evaluation of the searcher components, and carried out all experimental runs and evaluations