SEUPD@CLEF: Team BASETTE at LongEval: IR System for Basic Hardware*

Notebook for the LongEval Lab at CLEF 2025

Alberto Bottari[1], Lorenzo Croce[1], Fatemeh Mahvari Habib Abadi[1] and Nicola Ferro[1,*]

[1]Department of Information Engineering, University of Padua, Via Gradenigo 6/B, 35131 Padova, Italy

Abstract

This report describes the system developed by Team BASETTE for the CLEF 2025 LongEval Lab, Task 1 - Web Retrieval. Our main priority was optimizing performance on limited and commonly available hardware, deliberately avoiding the use of GPUs or other specialized computational resources. The system relies on classical Information Retrieval techniques, and is designed to run both indexing and retrieval in a multithreaded fashion to ensure high execution speed. During development, we explored various strategies, some of which were discarded not only due to limited effectiveness, but also because their processing time was not compatible with our efficiency constraints.

Keywords

Information Retrieval, CLEF 2025, LongEval, Web Search, Resource-Constrained Systems, Classical IR Techniques, Multithreaded Retrieval

1. Introduction

Information Retrieval (IR) systems play a fundamental role in enabling access to relevant data across massive, often heterogeneous corpora. As the volume of digital content continues to grow, the effectiveness and adaptability of IR systems are becoming increasingly important. From search engines and recommendation platforms to digital libraries and research databases, IR technologies underpin a wide array of applications. However, these systems are often sensitive to temporal drift in data and variation in document structures and query formulations, which motivates the need for robust retrieval architectures.

In the context of the LongEval Lab at CLEF [1], which focuses on evaluating the long-term effectiveness of retrieval systems, we have developed a configurable IR system built in Java and powered by Apache Lucene [2]. Our system supports a wide range of preprocessing, parsing, analysis, and indexing strategies through an extensible configuration, enabling us to experiment with different setups.

This paper presents the architecture, implementation, and features of our IR system.

The remainder of this paper is organized as follows: Section 2 describes the methodology and system architecture; Section 3 outlines the experimental setup; Section 4 presents our training work; Section 5 describes the performance enhancement after implementing each feature; and Section 6 concludes the report with a summary and suggestions for future improvements.

2. Methodology

In the following subsections, we provide a detailed breakdown of our methodology. We begin by describing the architecture of a classical IR system, detailing each of its main modules: parser, preprocessor,

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

*This manuscript follows the official CEUR-WS template.

*Corresponding author.

alberto.bottari@studenti.unipd.it (A. Bottari); lorenzo.croce.1@studenti.unipd.it (L. Croce); fatemeh.mahvarihabibabadi@studenti.unipd.it (F. Mahvari Habib Abadi); nicola.ferro@unipd.it (N. Ferro)

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

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

analyzer, indexer, and searcher. Next, we introduce our multithreading strategy, which maximizes performance under constrained hardware conditions. We then discuss how we leveraged automated hyperparameter optimization using Optuna to fine-tune our system. Subsequently, we highlight several approaches that were explored but ultimately discarded, either due to insufficient effectiveness or excessive computational cost. Finally, we explain how the system's configuration works.

At the core of the system is the `InformationRetrievalSystem` class, which integrates all major components of the pipeline, from preprocessing and parsing to indexing and searching, coordinating their interaction throughout the retrieval process. The system is launched via the `Main` class, which handles command-line argument parsing and initializes the configuration environment.

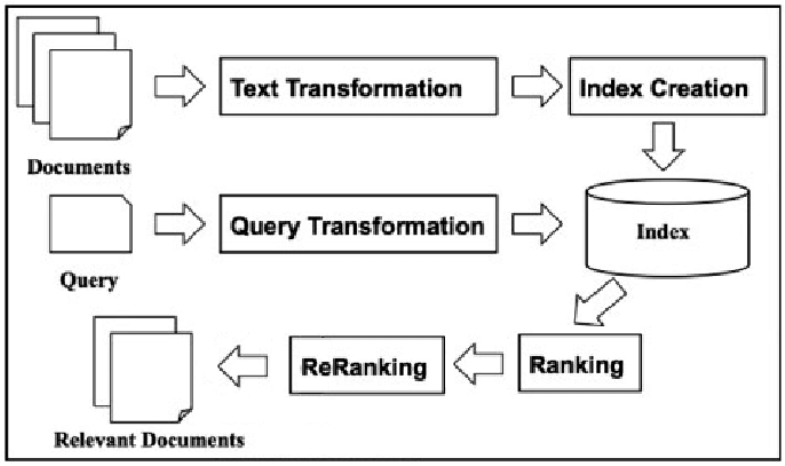

Figure 1: Architecture of our IR system.

2.1. Document Parsing

The document parsing process involves specialized parsers such as the `DirectoryDocumentParser` and `JsonFileDocumentParser`. The `DirectoryDocumentParser` recursively scans directories, processing each file as an individual document and filtering out unwanted files using user-defined patterns. The `JsonFileDocumentParser` handles JSON files containing arrays of documents, converting them into a structured format suitable for indexing. The parsed documents are converted into `ParsedDocument` objects, containing content and metadata for indexing. This approach allows for extendible integration of new parsers as needed, supporting various document formats and sources.

2.2. Query Parsing

Query parsing is the stage where raw query inputs are read, interpreted, and transformed into structured query objects suitable for execution by the search engine. This functionality is abstracted by the `IQueryParser` interface and implemented in one or more concrete classes. Each query is converted into a Lucene `Query` object and wrapped with metadata for tracking and scoring. This design allows the parser to be used in iterator-style loops and ensures that both the structured query and its metadata (e.g., query ID and original text) are available to downstream components. The primary implementation of `IQueryParser` in the system is `TxtQueryParser`. It reads a plain text file where each line contains one query, usually in the format:

```
<query-id>\t<query-text>
```

The `RawQuery` class encapsulates metadata associated with a query. It typically includes the `qid`, which is the unique identifier of the query, and the `text`, which is the raw textual form of the query as provided in the input file. The query text is then tokenized and processed using the configured analyzer,

and a corresponding Lucene `Query` object is generated. The parsed query text may be used to construct various types of Lucene queries, including term queries, phrase queries, and boolean queries. The final Lucene `Query` object is passed to the searcher.

2.3. Text Preprocessing

The system provides a mechanism to apply a chain of transformations to input queries or documents before tokenization. This is achieved using the `IPreProcessor` interface, which defines a single `process(String query)` method. Two notable implementations include the `UnicodeNormalizer PreProcessor`, which applies Unicode normalization, and the `RegexPreProcessor`, which applies regular expression substitutions using a predefined pattern and replacement string.

2.4. Text Analysis

The text analysis phase transforms raw textual input into a stream of tokens that can be indexed and later retrieved. The analysis is performed through the `GenericAnalyzer` class, the main analyzer component, which is dynamically assembled using configuration files specified via implementations of the `IAnalyzerConfig` interface. `GenericAnalyzer` follows a pipeline composed of:

- **Tokenizer**: Responsible for splitting the input text into raw tokens.
- **Token Filters**: Sequentially modify, normalize, or enrich the token stream.
- **Stemmer**: This component is in charge of removing suffixes and word endings. Even if in Lucene it's implemented as a token filter the importance of this component is high so we decided to make it explicitly configurable.

2.4.1. Tokenizer Configurations

Our system supports three tokenizer configurations, each defining how the input text is split into tokens:

- `StandardTokenizerConfig`: Uses Lucene's standard tokenizer, which follows Unicode Text Segmentation rules. It handles punctuation, digits, and word boundaries in a language-aware manner.
- `LetterTokenizerConfig`: Emits tokens consisting only of alphabetic characters. Any non-letter character (e.g., digits, punctuation) is treated as a delimiter.
- `WhitespaceTokenizerConfig`: Splits the input text strictly on whitespace characters. It does not handle or remove punctuation or symbols, those are retained as part of the tokens.

2.4.2. Token Filters

We have utilized Token Filters to expand and modify the token streams of both queries and documents. These filters are categorized into agnostic and language-specific types, and they are applied after tokenization.

Agnostic Token Filters These filters are language-independent and perform general text normalization or enhancement operations:

- `LowerCaseFilterConfig`: Converts all tokens to lowercase, ensuring case-insensitive search.
- `ASCIIFoldingFilterConfig`: Transforms accented and special characters into their closest ASCII equivalents.
- `TrimFilterConfig`: Removes any leading or trailing whitespace around each token.
- `LengthFilterConfig`: Discards tokens whose characters count is not in a certain (configurable) range .

- `RegexFilterConfig`: Uses regular expressions to remove or modify tokens based on pattern matching, for example html tags or not Unicode characters.
- `SpellCheckerFilterConfig`: Applies Lucene's spell-checking module to suggest corrections for tokens using the index.

Language-Specific Token Filters These filters handle language-specific linguistic features and are designed for either English or French:

- `FrenchStopFilterConfig`: Removes common French stopwords such as "le", "de", and "et".
- `FrenchElisionFilterConfig`: Removes French elisions in contractions (e.g., "l'homme" becomes "homme").
- `EnglishStopFilterConfig`: Removes frequent English stopwords such as "the", "and", and "of".
- `EnglishPossessiveFilterConfig`: Removes possessive suffixes (e.g., "company's" becomes "company").

2.4.3. Stemming

Stemming is handled via Lucene's built-in filters and configured directly within the analyzer. For English, the system supports minimal stemming, which reduces words to their root forms (e.g., "running" to "run") while avoiding overly aggressive normalization that might merge distinct terms.

For French, the system offers a broader range of options. It supports minimal stemming for light normalization using the `FrenchMinimalStemFilter`, a more conservative approach through the `FrenchLightStemFilter`, and a more aggressive strategy using the `FrenchSnowballStemFilter` with the French Snowball algorithm.

2.4.4. Word N-grams

In our system, we initially applied Lucene's `ShingleFilter`, a `TokenFilter` that generates n-grams—sequences of adjacent tokens—to enhance textual representation. This approach aimed to improve recall by capturing short-range context and enabling the system to match not just isolated terms but also common word combinations and phrases [3]. We applied the `ShingleFilter` during both indexing and query processing: at indexing time, to enrich document representations by storing word n-grams; and at query time, to perform query expansion by appending n-gram variants to the original query terms.

However, this naive n-gram indexing method revealed significant performance issues. The number of generated n-grams grows combinatorially with document length, causing both storage overhead and severe slowdowns during indexing due to the bloated inverted index. To mitigate this inefficiency, we adopted a refined strategy that avoids indexing all possible n-grams explicitly.

Instead, the updated system uses Lucene's phrase query capabilities to dynamically generate n-gram matches at query time. Rather than storing n-grams in the index, we now tokenize and index only unigrams but construct `PhraseQuery` objects from the original query text when searching. This approach maintains the benefits of contextual matching while reducing index size and indexing time. By leveraging Lucene's efficient positional indexing and skipping the storage of redundant n-gram entries, we achieve comparable retrieval effectiveness with much better performance.

2.5. Parallel Computing

An essential way to improve the performance of a search engine is not only by optimizing how queries are executed but also by addressing the time required to index documents [4]. The efficiency of both the indexing process and the execution of user queries is critical, particularly as the volume of documents grows. In our search engine project, we realized that running numerous experiments with

significant hardware constraints required careful attention to the time and resources consumed in these operations. To mitigate performance bottlenecks, we focused on optimizing both the indexing and searching processes using parallelism. The following sections details how parallelism was implemented to streamline the indexing and searching processes.

2.6. Indexing

The indexing stage is responsible for transforming processed documents into an inverted index using Lucene. This index enables efficient retrieval of relevant documents in response to user queries. The indexing logic is encapsulated in the `Indexer` class, which is instantiated at runtime based on external configuration.

2.6.1. Indexer

The `Indexer` class serves as the core component responsible for the indexing workflow. Its main function is to iterate through all parsed documents, apply the configured analyzer to the document content, and write the resulting tokens into a Lucene index.

The indexer operates on a designated input directory and outputs to a target index directory. It is responsible for several tasks related to indexing, including initializing a Lucene `IndexWriter` with the appropriate configuration, processing each document using the provided analyzer, and storing fields such as the document ID and body content. Additionally, it records term positions and frequencies to support proximity queries and scoring models like BM25 [5]. The indexer also logs important statistics, including the number of files processed and the time taken for the indexing process.

2.6.2. Concurrent Indexing Process

In this project, parallelism in the indexing process is implemented using a producer-consumer model. The producer threads are responsible for parsing documents, while the consumer threads handle the indexing of these parsed documents into a Lucene index. Once a document is parsed into a `ParsedDocument` by a producer, it is placed into a shared `BlockingQueue`, where it will be consumed. For this parallel indexing system, we use an empirical ratio found to perform well on our machine: 2/5 of the threads are allocated to document parsing (producers), and 3/5 to document indexing (consumers).

After all documents are processed, a special `POISON` marker is used to signal the end of the input stream and gracefully terminate consumer threads.

During these experiments, we also identified that one of the most significant bottlenecks was the commit-to-disk phase during indexing. On systems equipped with sufficient RAM (at least 32 GB in our tests), we enabled an in-memory indexing mode where the index was never written to disk. Instead, it remained entirely in RAM throughout the process. This approach reduced the total indexing time by nearly half, enabling a much higher number of experiments within the same time window. The decision to use disk-based or in-memory indexing remains configurable through the system's JSON configuration.

2.6.3. Index Field Configuration

The indexer writes each document's content into a dedicated Lucene field, typically named `BODY_FIELD`. This field is configured to store term frequency and position information, which is used for ranked retrieval and phrase queries. Additionally, it can optionally store the original body text, which may be useful in downstream components such as rerankers.

2.7. Searching

The searching stage is responsible for executing user queries over the previously created Lucene index and retrieving the most relevant documents. This process is implemented through the `Searcher` class, which orchestrates multithreaded query execution and writes ranked results in TREC run format.

The `Searcher` class serves as the main retrieval engine and is instantiated with several components. These include a Lucene `IndexReader` for accessing the index, a similarity model (such as BM25) to score documents, and a query parser that generates Lucene queries from raw query text, as discussed in Section 2.2.

2.7.1. Concurrent Searching Process

Initially, the search process followed a producer-consumer architecture to parallelize query processing across multiple threads. The `QueryProducerTask` ran in a dedicated thread that parsed and converted queries into Lucene-compatible objects wrapped in `QueryWrapper` instances. These were pushed into a shared blocking queue. Multiple `QuerySearchTask` threads then consumed the queries, executed searches via Lucene's `IndexSearcher`, and handled result formatting and output. A poison-pill mechanism was employed to signal the end of the query stream.

However, as the complexity of query generation increased—particularly with the addition of temporal reasoning and semantic rewriting—the producer-consumer model introduced subtle concurrency bugs that were difficult to debug and resolve. To simplify the control flow and reduce synchronization overhead, we migrated to a single-task parallel model. In this design, a single orchestrator task internally handles both query parsing and execution, distributing work across threads without the need for an intermediate queue.

Despite the structural simplification, we observed no drop in performance. CPU utilization remained at 100% throughout the search process, confirming that all available threads were effectively leveraged. This refactoring improved maintainability while preserving high throughput during the query evaluation.

2.7.2. Similarity Model

The system uses Lucene's BM25 [6] implementation as the default similarity model. Key parameters such as k_1 (term frequency saturation) and b (length normalization) are configurable.

Additionally, the retrieval logic includes a configurable thresholding mechanism to filter out low-quality results. Although the system may retrieve up to a fixed number of documents (e.g., 100), it evaluates each result relative to the top-ranked document. If a candidate's score falls below a configurable percentage of the highest score (e.g., less than 50% of the top score), it is discarded. This helps eliminate documents that are technically within the result limit but are likely to be irrelevant, improving the overall precision of the final ranked list.

2.8. Hyperparameter Optimization with Optuna

To explore the space of possible configurations and improve the overall performance of our retrieval system, we used *Optuna* [7], a Python library for hyperparameter optimization designed to automatically search for good parameter combinations. We chose it because it is easy to use, highly customizable, and fits naturally into our architecture where the system behavior is driven by a set of JSON configuration files.

In our setup, visible in figure 2, we treat each run of the search engine as a trial. For every trial, Optuna suggests a combination of parameters that define how the system behaves during indexing and search. These include:

- `BM25_PARAM_k1` → controls term frequency saturation in BM25.
- `BM25_PARAM_b` → controls length normalization in BM25.
- `STEM_FILTER_TYPE` → the type of stemming applied during indexing (e.g., `french_minimal`).
- `STOP_FILTER_TYPE` → stopword filtering strategy (e.g., built-in or generic).
- `STOP_FILTER_FILEPATH` → external stopword list file used if applicable.
- `LENGTH_MIN_LENGTH` → minimum term length allowed in the index.
- `LENGTH_MAX_LENGTH` → maximum term length allowed in the index.

- TEMPORAL_BOOST → weight applied to the temporal similarity score.
- NGRAM_BOOST → weight applied to the n-gram matching score.
- SCORE_THRESHOLD → minimum score a document must have to be considered relevant.
- MIN_NGRAM_SIZE → minimum size of n-grams used in the query expansion.
- MAX_NGRAM_SIZE → maximum size of n-grams used in the query expansion.
- MAX_DOC_RETRIEVED → maximum number of documents retrieved per query.
- PROXIMITY_RERANKER_MAX_SLOP → maximum distance 2 words can have for the proximity reranking.
- SPELLCHECKER_NUMBER_OF_SUGGESTIONS → number of spell correction suggestions considered.
- SPELLCHECKER_MIN_TERM_LENGTH → minimum length of a word to be spell-checked.
- BM25_WEIGHT → the weight of the bm25 for document retrieval
- PROXIMITY_WEIGHT → the weight of proximity search for document retrieval.

These parameters are inserted into two template files, `indexer.json` and `searcher.json`, replacing placeholder values.

Once the JSON files are generated, the system launches the search engine with these configurations across snapshots. After each run, the results are evaluated using `trec_eval`, and a custom score is computed based on the output. Optuna also takes care of managing the history of all experiments: it keeps track of which parameter combinations have been tried, how well they performed, and which trial produced the best result so far. More importantly, it supports an early stopping mechanism called *pruning*, which allows the system to interrupt trials that appear unpromising before they complete all evaluations.

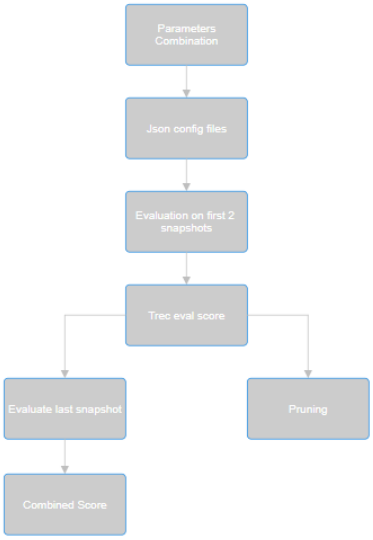

Figure 2: Sum of optuna's script workflow.

To address this, we compute a custom score that combines multiple aspects of performance. First, we calculate the average nDCG over the document snapshots evaluated so far, to capture overall effectiveness. Then, we assess the relative drop in performance between consecutive snapshots to penalize instability over time. These components are combined into a single score that rewards both high effectiveness and stable behavior.

Formally, the score S used during optimization is defined as:

$$S = \lambda \cdot \text{mean}(nDCG) - (1 - \lambda) \cdot \text{std}(nDCG)$$

where $\lambda \in [0, 1]$ is a tunable parameter (configured via the `.env` file) that controls the trade-off between average effectiveness and result stability.

As the optimization proceeds, Optuna continuously explores the parameter space and learns which configurations are more effective. The figure 3 shows the progression of average nDCG values across trials, highlighting how the system gradually converges toward more effective and stable configurations. It's worth noting that, before the system reaches its optimal and stable state, there is an intermediate period of instability, likely caused by Optuna's extensive exploration of the parameter space.

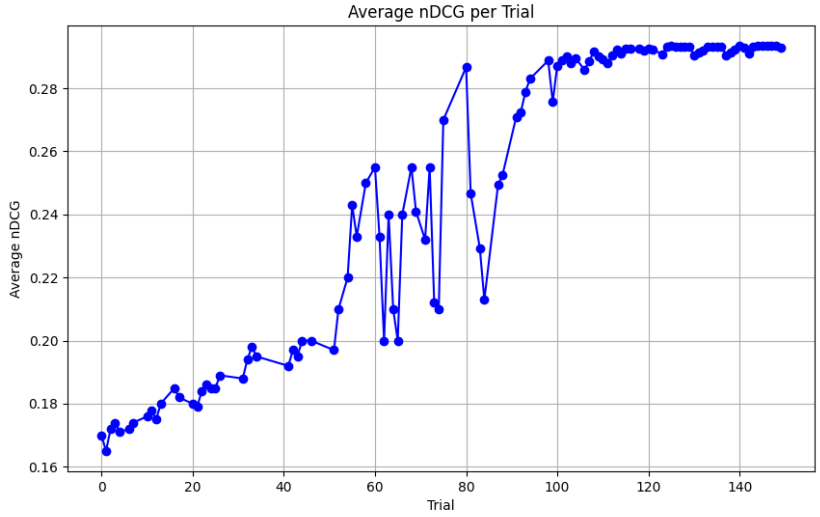

Figure 3: ndcg average value's evolution during optuna's trials.

2.9. Discarded Approaches

Throughout the development process, we investigated several techniques that initially appeared promising for improving system performance. However, due to technical limitations, inadequate lexical coverage, or integration complexity, some of these approaches did not yield the expected results. The following sections briefly describe these attempts and the reasons why they were ultimately abandoned.

2.9.1. Semantic Expansion with WordNet

To enhance semantic understanding during query processing, we explored the use of *WordNet* [8], a large lexical database of English developed at Princeton University. WordNet organizes nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms (synsets), each expressing a distinct concept. Its network structure of semantic relationships—such as synonymy, hypernymy, and meronymy—makes it a valuable resource for query expansion and contextual matching in information retrieval systems.

Goal and Integration Attempt Our goal was to use WordNet to expand query terms with semantically related alternatives (e.g., synonyms, hypernyms), ideally improving recall for queries that might otherwise be too narrow or under-specified. Integration into our Lucene-based system was envisioned through a token filter pipeline that would enrich tokens at query time using synsets retrieved from WordNet. We implemented a `WordnetTokenFilterConfig` component and began testing this expansion strategy.

To access WordNet programmatically, we relied on `extJWNL` [9], a Java API that facilitates interaction with WordNet dictionaries and synsets. It provided a convenient way to retrieve semantic relations for English terms and integrate them into the Lucene pipeline.

However, our dataset was predominantly in French, so we sought a compatible lexical resource in that language. The most promising candidate was *WOLF* (WordNet Libre du Français) [10], an open-source French WordNet project built to mirror the structure of the original Princeton WordNet.

Limitations and Abandonment Unfortunately, WOLF posed several challenges. First, its format was not compatible with the most common Java-based WordNet access libraries such as `extJWNL`. Adapting the data for these tools would have required extensive manual transformation or building custom parsers. Furthermore, WOLF's coverage and structural consistency were limited compared to the English WordNet. Many French terms lacked synsets or had only shallow hierarchies of related concepts.

After multiple attempts to preprocess and align the WOLF dataset to our tokenizer and filter pipeline, we were unable to achieve a working integration. No synonym expansion was successfully applied in practice.

Conclusion Given these technical barriers and the lack of tooling for French WordNet resources, we ultimately decided to abandon this route. Nonetheless, we believe that WordNet-based semantic expansion remains a promising strategy for IR, particularly in monolingual English scenarios or when higher-quality lexical ontologies are available for the target language.

2.9.2. Query Expansion via Synonym Dictionaries

In an attempt to improve recall on short or under-specified queries, we experimented with a query expansion strategy based on synonym substitution. The idea was to extend queries with semantically related terms, increasing the likelihood of retrieving documents that used alternative wordings. This approach is well-established in information retrieval, particularly when dealing with sparse queries or lexical variability between user language and document language.

Dictionary-Based Expansion To implement this idea, we adopted an open-source synonym dictionary from a GitHub repository [11]. Specifically, we extracted and converted the core data contained in the `dictionary.go` file, which maps base terms to lists of synonyms. We reformatted this data into a format compatible with Lucene's `SynonymFilter`, allowing integration into our analyzer pipeline.

The expansion logic was applied selectively: if a query was deemed too short (e.g., less than a configurable number of tokens), we automatically augmented it by adding one synonym per word, based on dictionary availability. If the query remained short even after the first pass, additional rounds of expansion were performed iteratively, appending further synonyms where possible.

Evaluation and Limitations Although this strategy was theoretically sound and straightforward to implement, it failed to produce the expected improvements in retrieval quality. Upon closer inspection, we found that the synonym pairs in the dictionary were often weakly related or even misleading. Many base terms had no meaningful synonyms, and in other cases the listed synonyms introduced semantic drift, retrieving documents that were topically unrelated to the original query intent.

Rather than boosting recall, the expansion often diluted query specificity, resulting in lower precision. This outcome was disappointing, especially considering the added complexity and overhead introduced in the query processing pipeline.

Conclusion We ultimately decided to abandon this synonym-based expansion strategy. However, our conclusion is not a rejection of the approach itself, but rather a reflection of the limited quality of the available lexical resource. We remain confident that, given a richer and more context-aware synonym dictionary, ideally one derived from a semantic model or manually curated lexicon, query expansion could be an effective tool in improving retrieval performance.

2.9.3. Date Normalization with Duckling

To handle temporal information in both documents and queries, we initially integrated *Duckling* [12], a library for recognizing and normalizing date expressions. The goal was to better align documents and queries referring to the same time periods, even if expressed differently (e.g., last summer" vs. July 2022").

Duckling, developed by Facebook AI Research, is an open-source system designed to extract structured data such as dates, durations, numbers, and quantities from natural language text. One of its key advantages is its multilingual capability, including support for English and French, which made it attractive for our multilingual setup. Another important factor in its initial selection was its availability as a self-contained, Dockerized microservice [13], allowing easy integration into our Java-based system via HTTP APIs returning structured JSON.

Architecture and Optimization Our first implementation used a microservice architecture: Duckling ran locally as a Docker container exposing an HTTP API. During indexing, each document was sent individually to Duckling, which returned all detected temporal expressions. From these, we computed the *median timestamp* and stored it in a dedicated Lucene field. At query time, the same process was applied: queries were passed to Duckling, and if temporal expressions were found, the resulting normalized timestamp was used to boost matching documents occurring around the same time.

Although conceptually elegant, this solution rapidly became a major performance bottleneck, particularly during indexing, where millions of HTTP calls to Duckling introduced significant overhead. To address this, we moved date extraction to a *preprocessing step*. We extracted timestamps in batch mode and built a map of `doc_id` $\rightarrow$ `median_timestamp`, which was then passed to the indexer. This drastically improved performance and allowed timestamp reuse across runs, as long as the documents remained unchanged.

Instability and Abandonment Despite these optimizations, we encountered stability issues. In particular, some queries containing a high density of temporal expressions on very closely spaced date mentions would cause Duckling to crash, terminating the processing thread. After inspecting the problem, we observed that Duckling consistently failed when parsing certain complex queries with many adjacent or overlapping temporal entities.

We attempted to bypass the Docker abstraction and compiled Duckling natively from source using Haskell but the crashes persisted. At this point, we conducted a statistical analysis of the dataset and found that only approximately 2% of all queries actually contained temporal expressions. Given this low impact on the overall retrieval process and the persistent technical issues, we concluded that Duckling's inclusion did not justify its cost in terms of system complexity and stability.

Consequently, we decided to permanently abandon Duckling from our system even if the concept of temporal alignment still remains a possible relevant solution in a IR system.

2.9.4. Neural Reranking with CamemBERT

In an effort to improve result ranking quality beyond traditional lexical matching, we experimented with a reranking phase based on *CamemBERT* [14]. CamemBERT is a transformer-based language model specifically trained for the French language. It is based on the RoBERTa architecture and was trained on a 138GB corpus composed of high-quality French text sources, including OSCAR, CCNet, and Wikipedia. Its monolingual training makes it particularly well-suited for semantic understanding in French retrieval tasks.

Usage and Implementation We employed CamemBERT to rerank the top-k documents retrieved via BM25. For each query-document pair, embeddings were computed and cosine similarity was used to reorder documents by semantic proximity.

To run the model within our Java-based search system, we used the `Deep Java Library (DJL)` [15], a high-level, engine-agnostic deep learning framework for Java. DJL allows direct integration of pre-trained neural models into Java applications, and supports backends like PyTorch, TensorFlow, and ONNX. However, DJL expects models to be in the PyTorch `.pt` format for PyTorch-based inference.

Since the official CamemBERT model available from Hugging Face [16] is provided in the Transformers format (with separate configuration and weights files), we had to download it manually and convert it into a serialized `.pt` format compatible with DJL. A Python script using the `transformers` and `torch` libraries handled the conversion and model tracing, enabling us to load it directly via DJL at runtime.

Performance Limitations and Abandonment While the reranking phase showed improvements in output quality, particularly when initial BM25 scores failed to capture deeper semantic relevance, the computational overhead was prohibitive. In the absence of a GPU, all inference operations ran on CPU, resulting in severe performance degradation. Reranking even a small batch of candidate documents (e.g., top-10) per query led to a time increase of approximately 8000–10000% compared to the baseline BM25-only pipeline. If we sum this with the fact that a single snapshot consisted of approximately of 75,000 queries we concluded that under these conditions the total runtime for a full evaluation became unmanageable.

Given that our system was designed with performance and large-scale tunability in mind, we concluded that this approach was not viable within our project's constraints. Despite its potential benefits in terms of ranking quality, the cost in execution time was too high, and we ultimately decided to discard the BERT-based reranking phase from the final version of the system.

We plan to revisit this direction in the future under better hardware conditions, as transformer-based reranking remains one of the most effective techniques for semantic matching.

2.10. System Configurability

A key design goal of our system was to ensure that all behavior-critical decisions could be modified without altering the Java source code. To this end, we encapsulated every tunable subsystem within serializable configuration object, organized under the `de.ir_lab.longeval25.config` package hierarchy.

Each configuration object implements a corresponding factory interface and is annotated to support deserialization via Jackson [17]. This allows the appropriate Java classes to be instantiated at runtime based solely on a `type` field in a JSON file.

At runtime, the main driver reads a user-specified configuration file (e.g., `searcher.json`) from the folder provided via the command line. This JSON file is then parsed using a shared `ObjectMapper`, which deserializes it into the corresponding configuration object.

During deserialization, runtime dependencies, such as command-line `Parameters`, are injected automatically using the `@JacksonInject` annotation. Once the configuration object is fully constructed, its `toRuntime()` method is called to produce the actual working component, ready for execution.

This level of configurability was fundamental to our workflow, as it enabled integration with automatic hyperparameter tuning tools like Optuna. Thanks to the JSON-based setup, experiments could be rapidly iterated without any need for recompilation or repackaging of the system. For example, switching from one type of stopword list to another (e.g., from a minimal to an aggressive set) requires only a one-line change in the configuration JSON file.

3. Experimental Setup

The experimental conditions under which our information retrieval system was developed, deployed, and evaluated are as follows:

- **Used Collections:** All experiments were conducted using the LongEval CLEF 2025 Lab dataset [18], which included documents in various formats (such as `.json`,`.trec`, etc.), queries in plain text format, and the corresponding relevance judgement files.
- **Evaluation Measures:** System performance was assessed using standard information retrieval metrics included in the `trec_eval` tool.
- **Source Code Repository:** The full source code is available at: https://bitbucket.org/upd-dei-stud-prj/seupd2425-basette
- **Hardware Used for Experiments:** All experiments were executed on a high-performance server with the following specifications:
  - CPU: AMD Ryzen 9 5950X (16 cores, 32 threads)
  - RAM: 128 GB DDR4
  - Storage: 2 × 4 TB NVMe SSDs
  - GPU: Not utilized
  - Operating System: Ubuntu 24.04.2 LTS
- **Software Environment:** The system was run with the following environment:
  - Java 17 (OpenJDK)
  - Maven 3.8.6
  - Python 3.11 (for document formatting utilities)
  - Lucene 9.0.0

  A Docker container was used to hold both the IR system and an image of a LaTeX library for compiling this report.
- **Run Procedure:** For detailed information about the steps required to run the project please refer to the readme.md file in the BitBucket repository.

4. Training

In this section, we present and analyze the results from various experiments conducted to optimize the performance of our system. A primary focus of these experiments was to identify the best configuration of system components by adjusting various hyperparameters. The process of fine-tuning these parameters, specifically using Optuna, allowed us to explore the vast parameter space and determine the most effective settings for our retrieval system.

The system architecture has distinct configurations for both the indexer and the searcher. Both configurations include placeholders for the hyperparameters, which Optuna dynamically selects to optimize the performance of the system. The results of these tests are presented in the following sections.

For the purposes of training our system we used the CLEF‘s collections snapshots visible in table 1 The results and plots presented in the following sections are summarized in the tables 2 3. For clarity

Table 1
CLEF’s collections used in the tests.

Parts of the CLEF collection used for testing
2022-07_fr
2022-08_fr
2022-09_fr
2022-10_fr
2022-11_fr
2022-12_fr
2023-01_fr
2023-02_fr

and conciseness, highly similar configurations have been omitted.

Table 2
Params and metrics results within trials - Part 1: Configuration Parameters.

LEN_MAX	LEN_MIN	NGRAM_MAX	NGRAM_MIN	NGRAM_BOOST	SCORE_THR	STEM_TYPE
14	2	3	3	1.46	0.54	french_minimal
15	2	2	2	2.939	0.749	french_light
7	1	2	2	2.368	0.632	french_light
8	2	5	3	2.085	0.542	french_light
14	3	3	2	1.127	0.593	french_snowball
17	3	3	2	1.013	0.653	french_snowball
11	2	5	2	1.484	0.701	french_light
20	1	4	3	1.770	0.790	french_minimal
19	1	4	3	1.739	0.776	french_minimal
12	1	4	3	1.922	0.599	french_minimal
12	1	4	3	1.881	0.589	french_minimal
16	2	4	3	1.220	0.569	french_minimal
17	2	3	3	1.238	0.558	french_minimal
16	3	4	3	1.285	0.569	french_minimal
16	3	4	3	1.050	0.564	french_minimal
13	2	4	3	1.003	0.530	french_minimal
13	3	4	3	1.009	0.619	french_minimal
18	2	4	3	1.185	0.659	french_minimal
20	2	4	3	1.178	0.664	french_minimal
18	2	4	3	1.322	0.681	french_minimal
16	3	4	3	1.009	0.567	french_minimal
15	2	4	3	1.125	0.574	french_minimal
14	2	4	3	2.855	0.638	french_minimal
15	3	4	3	1.115	0.547	french_light
15	2	4	3	2.773	0.635	french_minimal
15	2	4	3	2.805	0.552	french_minimal
15	2	4	3	2.780	0.558	french_minimal
15	2	4	3	2.767	0.555	french_minimal
15	2	2	2	2.653	0.521	french_minimal
15	2	3	2	2.713	0.559	french_minimal
15	2	2	2	2.723	0.557	french_minimal
17	2	2	2	2.431	0.539	french_minimal
17	2	2	2	2.462	0.539	french_minimal
19	2	2	2	2.485	0.523	french_minimal
20	2	2	2	2.911	0.523	french_snowball
17	2	3	2	2.656	0.511	french_light
17	2	3	2	2.685	0.512	french_light
17	2	3	2	2.675	0.505	french_light
18	2	3	2	2.652	0.511	french_light
18	2	3	2	2.646	0.509	french_light
18	2	3	2	2.664	0.508	french_light
18	2	3	2	2.354	0.507	french_light
18	2	3	2	2.881	0.502	french_light
20	2	3	2	2.554	0.514	french_light
18	2	3	2	2.691	0.530	french_light
18	2	3	2	2.670	0.500	french_light
18	2	3	2	2.648	0.508	french_light
19	2	3	2	2.536	0.501	french_light
20	2	3	2	2.556	0.531	french_light
17	2	3	2	2.838	0.533	french_light
18	2	3	2	2.686	0.502	french_light
17	2	3	2	2.633	0.520	french_light
16	2	3	2	2.620	0.518	french_light
17	2	3	2	2.536	0.530	french_light
16	2	3	2	2.812	0.518	french_light

Table 3
Params and metrics results within trials - Part 2: Additional Parameters and Results.

STOP_FILE	k1	b	PROX_W	BM25_W	MAX_SLOP	SPELL_N	SPELL_L	Score	nDCG
lang-resource-stopwords.txt	1.77	0.78	0.132	0.859	0	4	7	0.194	0.286
None	0.938	0.366	0.912	0.104	1	1	6	0.168	0.246
stopwords-filter-fr.txt	0.777	0.969	0.200	0.787	0	2	8	0.095	0.139
fergiemcdowall_stopwords_fr.txt	1.656	0.074	0.814	0.132	3	0	5	0.131	0.193
None	1.656	0.493	0.713	0.245	1	3	4	0.154	0.225
None	1.844	0.318	0.752	0.178	5	4	4	0.169	0.249
ranksnl-french.txt	1.957	0.962	0.234	0.112	8	2	1	0.171	0.252
lang-resource-stopwords.txt	0.981	0.186	0.655	0.345	7	2	0	0.183	0.271
lang-resource-stopwords.txt	0.886	0.659	0.677	0.333	6	1	1	0.184	0.272
lang-resource-stopwords.txt	1.329	0.738	0.701	0.298	5	0	2	0.189	0.278
lang-resource-stopwords.txt	1.329	0.727	0.552	0.437	5	1	3	0.192	0.283
bbalet_stopwords_fr.txt	1.299	0.743	0.603	0.411	6	1	4	0.196	0.288
bbalet_stopwords_fr.txt	1.546	0.855	0.678	0.236	4	2	3	0.187	0.275
bbalet_stopwords_fr.txt	1.621	0.861	0.712	0.258	5	2	5	0.194	0.287
bbalet_stopwords_fr.txt	1.524	0.579	0.834	0.103	4	1	4	0.195	0.289
bbalet_stopwords_fr.txt	1.907	0.824	0.592	0.274	3	5	5	0.196	0.290
bbalet_stopwords_fr.txt	0.522	0.506	0.756	0.219	2	2	4	0.195	0.288
bbalet_stopwords_fr.txt	1.985	0.846	0.767	0.285	4	1	3	0.196	0.289
bbalet_stopwords_fr.txt	1.818	0.637	0.698	0.142	4	2	4	0.193	0.285
None	1.163	0.993	0.812	0.137	4	2	5	0.195	0.288
bbalet_stopwords_fr.txt	1.146	0.982	0.534	0.496	1	1	4	0.197	0.291
bbalet_stopwords_fr.txt	1.160	0.867	0.723	0.208	3	0	4	0.196	0.290
bbalet_stopwords_fr.txt	1.099	0.902	0.640	0.360	5	3	5	0.196	0.289
geonetwork-fre.txt	1.432	0.907	0.789	0.131	4	1	8	0.195	0.287
bbalet_stopwords_fr.txt	1.418	0.802	0.456	0.579	4	1	7	0.196	0.290
bbalet_stopwords_fr.txt	1.467	0.897	0.678	0.324	2	2	5	0.198	0.292
bbalet_stopwords_fr.txt	1.063	0.665	0.712	0.251	3	2	4	0.197	0.291
stopwords-filter-fr.txt	1.242	0.916	0.834	0.159	4	2	5	0.198	0.292
None	1.778	0.585	0.592	0.402	1	1	6	0.198	0.292
None	1.409	0.791	0.726	0.268	4	2	5	0.198	0.292
None	1.198	0.931	0.467	0.415	2	1	4	0.198	0.292
None	1.217	0.829	0.698	0.301	3	2	4	0.198	0.292
None	1.228	0.934	0.812	0.186	5	1	3	0.198	0.292
None	1.023	0.757	0.534	0.449	6	0	4	0.198	0.292
None	0.820	0.896	0.723	0.318	4	3	2	0.197	0.290
None	1.376	0.705	0.645	0.393	2	1	3	0.199	0.293
None	1.220	0.370	0.789	0.265	4	4	5	0.199	0.293
None	0.620	0.810	0.456	0.427	5	2	5	0.199	0.293
None	0.914	0.997	0.678	0.321	4	1	4	0.199	0.293
None	1.145	0.004	0.712	0.209	4	1	5	0.199	0.293
None	1.227	0.934	0.734	0.256	4	2	4	0.198	0.293
None	1.281	0.933	0.592	0.402	3	1	4	0.197	0.290
None	0.998	0.960	0.756	0.188	5	2	3	0.197	0.291
None	0.965	0.946	0.467	0.471	1	2	4	0.198	0.292
None	0.995	0.946	0.698	0.302	4	2	5	0.199	0.293
None	0.840	0.944	0.701	0.299	4	1	5	0.198	0.293
None	0.697	0.950	0.534	0.465	3	1	5	0.198	0.293
None	0.845	0.189	0.590	0.410	5	1	6	0.199	0.293
None	0.712	0.876	0.645	0.355	4	2	5	0.197	0.290
None	0.955	0.997	0.789	0.189	4	2	4	0.197	0.291
None	1.084	0.950	0.656	0.264	2	1	5	0.198	0.292
None	1.059	0.960	0.678	0.326	3	2	4	0.199	0.293
None	1.079	0.772	0.712	0.283	5	1	5	0.198	0.293
None	0.871	0.854	0.645	0.355	4	2	5	0.197	0.291

4.1. Indexing

Table 4
Indexing Performance on Document Dataset.

Indexing Approach	Document Dataset Size (GB)	Index Size (GB)	Average Time (s)
Single Thread	12	12	1800
Multithread	12	12	176

The performance results of our indexing process, as shown in Table 4, highlight the significant improvements achieved through multithreading. When using a single-threaded approach, indexing a 12 GB document dataset took approximately 1800 seconds, whereas with multithreading, the same task was completed in just 176 seconds, demonstrating a substantial reduction in indexing time.

4.2. Stemming Strategy

In our system, we evaluated three stemming strategies tailored for the French language: a conservative approach through the `FrenchMinimalStemFilter`, a more balanced method via the `FrenchLightStemFilter`, and a more aggressive strategy using the `FrenchSnowballStemFilter`, which is based on the Snowball algorithm.

As shown in Table 2, Optuna consistently selected the `FrenchLightStemFilter` as the most effective stemming strategy. It outperformed both the minimal and snowball approaches across the different evaluation metrics, improving retrieval performance without introducing the risks of under- or over-stemming commonly associated with the other two methods.

4.3. Stopword Filtering

Stopword filtering is a preprocessing step in text retrieval systems, aimed at removing high-frequency words that typically carry low semantic content.

In our setup, we compared nine different stopword lists for French, including several curated sets from the open-source repository `stopwords-iso` [19], which offers community-maintained stopword lists in multiple formats and dialectal variations.

Optuna selected the default French stopword list provided by Lucene, accessed via `FrenchAnalyzer.getDefaultStopSet()` [20], as the most effective configuration. This built-in list, integrated directly in the Apache Lucene library, appears to provide a balanced and domain-independent set of common French stopwords that generalizes well across document collections.

As shown in Table 3, this default set outperformed all alternative configurations in terms of retrieval performance, highlighting the reliability of the default Lucene stopword strategy when no domain-specific list is clearly superior.

4.4. BM25 Similarity Parameter Tuning

Tuning of Parameter `b` The parameter `b` in the BM25 similarity function controls the extent to which document length is normalized. A value of `b` close to 0 reduces the influence of document length, while a value close to 1 fully normalizes scores based on length differences between documents.

As shown in Figure 4, Optuna consistently converged toward values of `b` around 0.85 during the optimization process. This result aligns well with theoretical expectations in Information Retrieval literature, where `b` values between 0.75 and 0.9 are commonly found to perform well across diverse corpora [21]. In our case, this setting suggests that moderate document length normalization is beneficial for our collections, allowing longer documents to remain competitive without overwhelming shorter ones.

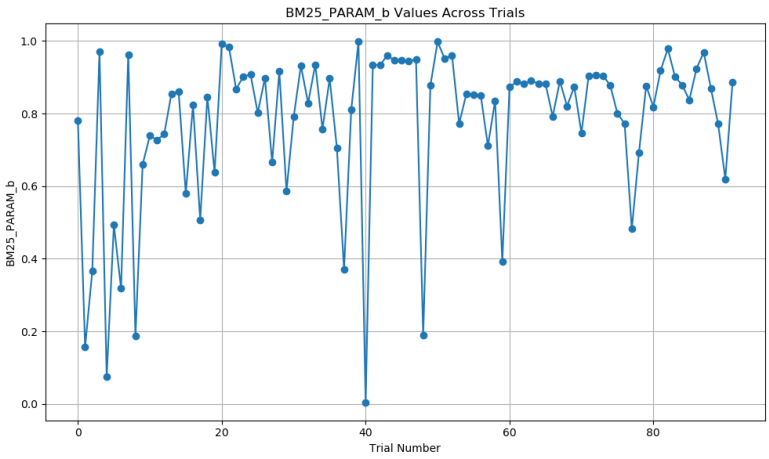

Figure 4: Evolution of BM25 b parameter values selected by Optuna over multiple trials.

Tuning of Parameter k_1 The parameter k_1 regulates the influence of term frequency in the BM25 scoring function. Higher values increase the contribution of frequently occurring terms in the document, typically leading to better discrimination between relevant and non-relevant documents.

Unexpectedly, as illustrated in Figure 5, Optuna identified an optimal value for k_1 around 0.95 – lower than the commonly cited optimal range of 1.2 to 2.0. This result might seem counterintuitive at first. One possible explanation lies in the nature of our collection and preprocessing pipeline: stemming, stopword filtering, and n-gram boosting might already amplify term frequency signals or reduce noise, making a lower k_1 sufficient or even preferable. Another contributing factor could be the presence of short, focused documents or queries, where overly aggressive TF scaling leads to overfitting on frequent terms.

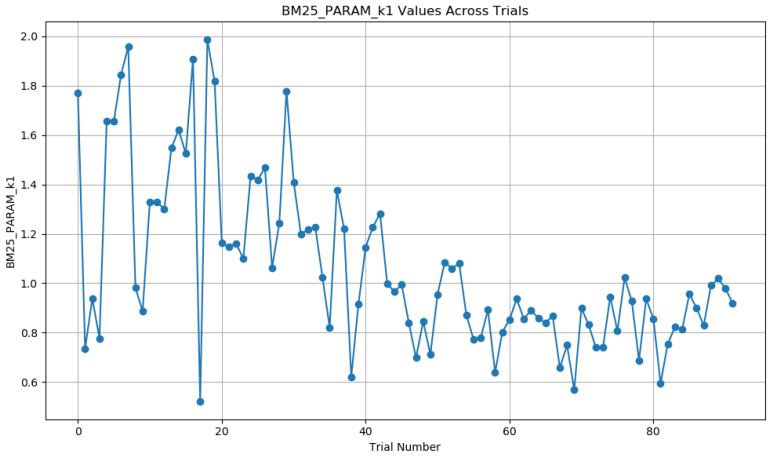

Figure 5: Evolution of BM25 k_1 parameter values selected by Optuna over multiple trials.

Impact of BM25 Parameters on Final Performance Ultimately, the BM25 parameters had a strong influence on the final retrieval quality, as BM25 remains the backbone of our ranking system. As shown in Figure 6, where lighter (yellowish) colors indicate higher performance values, the combination of $k_1 = 0.9$ and $b = 0.85$ emerged as the most effective configuration. These 3D graphs summarize the joint effect of both parameters across the collections.

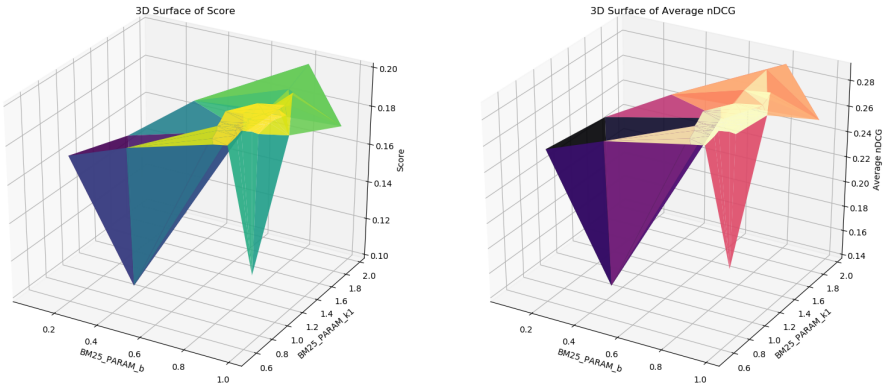

Figure 6: 3D surface plot showing average NDCG values and our custom score across combinations of k_1 and b.

4.5. Word N-grams

Word N-grams are contiguous sequences of n items, typically words, extracted from text, commonly used in natural language processing tasks to capture local contextual patterns. They offer a balance between simplicity and effectiveness, making them valuable for applications such as query expansion, auto-completion, spell correction, and ranking. In information retrieval, n-grams help identify multi-word expressions and improve matching between queries and documents.

Minimum Word N-gram Size The parameter `MIN_NGRAM_SIZE` controls the shortest sequence of tokens considered during indexing and query expansion. Lower values (e.g., 1 or 2) focus on very short phrases, which may increase recall but can introduce noise. As shown in Figure 7, Optuna identified that a minimum size of 2 yields the best performance. This setting provides a good compromise: it captures essential short phrases without being overly sensitive to isolated terms or subword tokens.

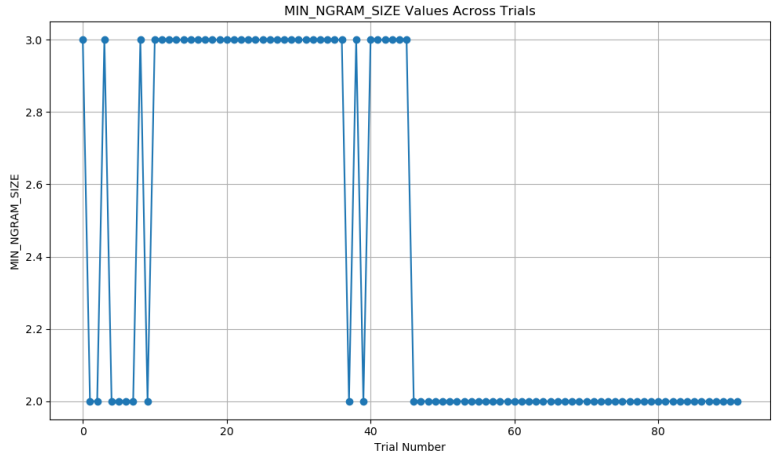

Figure 7: Effect of varying `MIN_NGRAM_SIZE` on performance during Optuna optimization trials.

Maximum N-gram Size The parameter `MAX_NGRAM_SIZE` sets the upper bound on the length of word n-grams to be used. Larger values can help detect meaningful multi-word expressions (e.g., "freedom of speech" or "climate change"), but they also increase the number of candidate terms, slowing down indexing and introducing sparsity. Figure 8 shows the behavior of this parameter across trials: Optuna found that a value of 3 consistently delivered strong results, capturing useful short phrases without the overhead of longer, rarer n-grams.

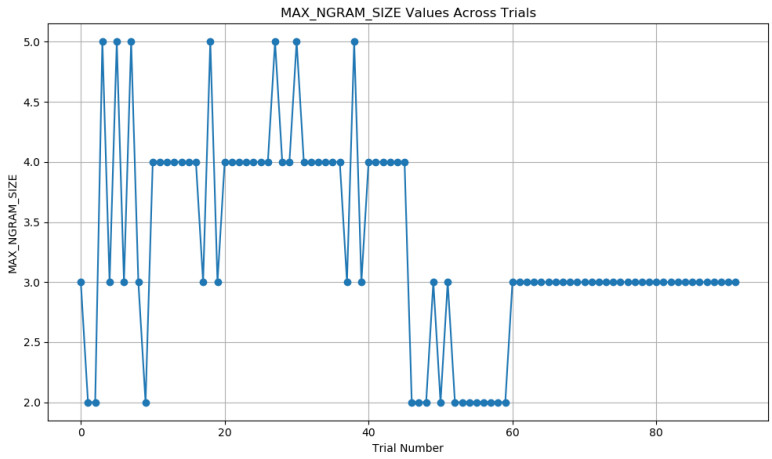

Figure 8: Effect of varying MAX_NGRAM_SIZE on performance during Optuna optimization trials.

Combined Effect on Precision and Recall To assess the combined effect of MIN_NGRAM_SIZE and MAX_NGRAM_SIZE, we analyzed the precision-recall behavior of the system under various configurations. Figure 9 presents the resulting PR curves from TREC evaluation. The optimal configuration, MIN_NGRAM_SIZE = 2 and MAX_NGRAM_SIZE = 3, emerges clearly as it yields the best balance between precision and recall.

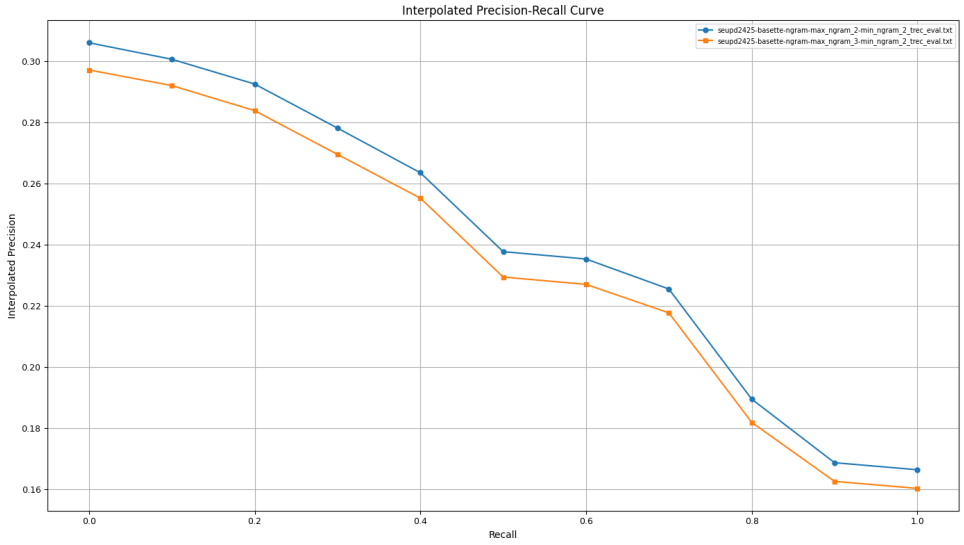

Figure 9: Precision-Recall curves for different combinations of n-gram parameters, showing the best result with MIN = 2, MAX = 3.

N-gram Boost in Query Scoring Beyond indexing, n-grams also influence the scoring phase through a dedicated boosting mechanism: when an n-gram extracted from the query matches an indexed phrase in a document, the score of that document is increased proportionally to a weight parameter called NGRAM_BOOST.

Optuna identified that a relatively high value, around 2.75, was optimal for this boost factor. This suggests that multi-word expressions play a significant role in distinguishing relevant documents within our collections. By giving extra weight to n-gram matches, the system prioritizes documents that preserve query intent more precisely.

This outcome also aligns with linguistic intuition: key phrases (e.g., "data protection act", "financial crisis") often carry more semantic value than individual terms.

As shown in Figure 10, Optuna trials converged toward this higher value, reinforcing the importance of phrase-level matching.

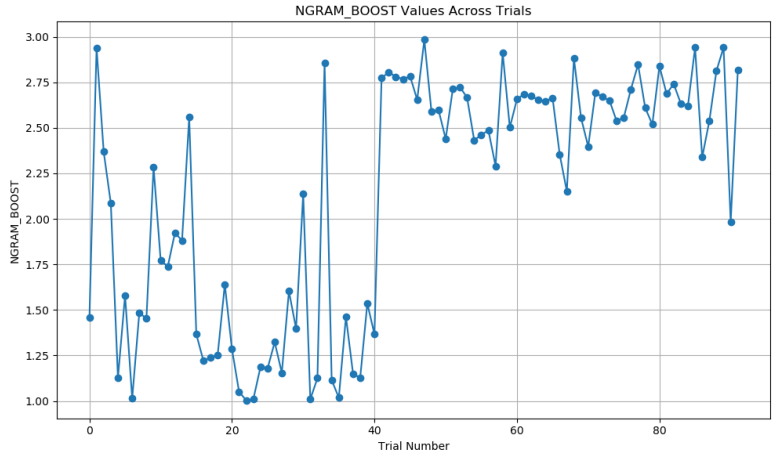

Figure 10: Evolution of NGRAM_BOOST values during Optuna optimization. The optimizer consistently favors high boosting weights around 2.75.

4.6. Length Filter

Length filtering is a fundamental preprocessing step in many Information Retrieval systems. Its goal is to eliminate tokens that are either too short or too long, which are often semantically uninformative or problematic for indexing.

Very short tokens (e.g., one or two characters) often correspond to punctuation, prepositions, articles, or incomplete terms resulting from tokenization errors. Including them increases index size without improving retrieval quality and can introduce noise during matching. On the other end of the spectrum, extremely long tokens are typically rare compound words, malformed terms, or noisy artifacts, especially in user-generated content, and may negatively affect performance due to low frequency and high indexing cost [22].

To address this, our system applies a *length filter* both during indexing and query processing, removing tokens outside a predefined length interval, defined by two parameters: LENGTH_MIN_LENGTH and LENGTH_MAX_LENGTH.

Minimum Token Length The parameter LENGTH_MIN_LENGTH sets the lower bound on the length of tokens to be retained. As shown in Figure 11, Optuna trials indicate that setting this value around 2 offers the best trade-off between eliminating noise and preserving useful query terms. This prevents overly generic tokens like "a", "on", or "at" from affecting the scoring.

Maximum Token Length The parameter LENGTH_MAX_LENGTH defines the upper bound of acceptable token length. Words that exceed this length are typically rare or malformed, and indexing them can result in unnecessary computational overhead. Figure 12 shows how Optuna explored this parameter, ultimately converging around a value of 17, which effectively removes outliers without discarding meaningful multi-word terms.

Joint Impact on Retrieval Performance The combined effect of the minimum and maximum token length parameters was analyzed using 3D visualizations of retrieval performance. Figure 13 illustrate how different combinations influence overall system effectiveness. These plots show that a configuration around LENGTH_MIN_LENGTH = 2 and LENGTH_MAX_LENGTH = 17 yields the best balance, filtering out irrelevant or noisy tokens while retaining informative ones.

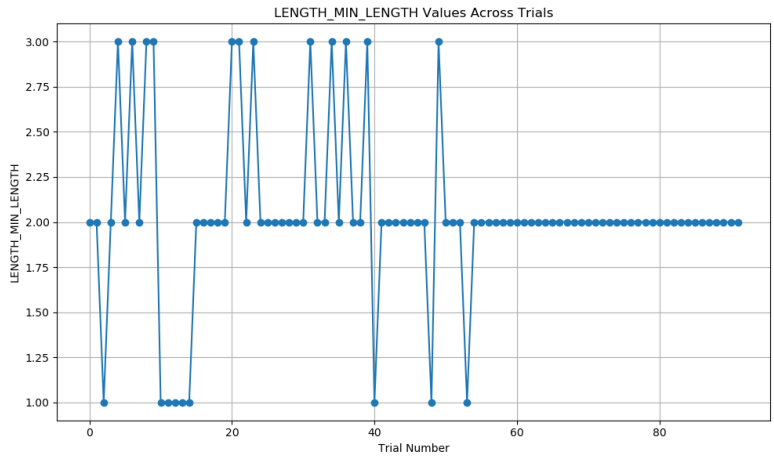

Figure 11: Evolution of LENGTH_MIN_LENGTH during Optuna optimization.

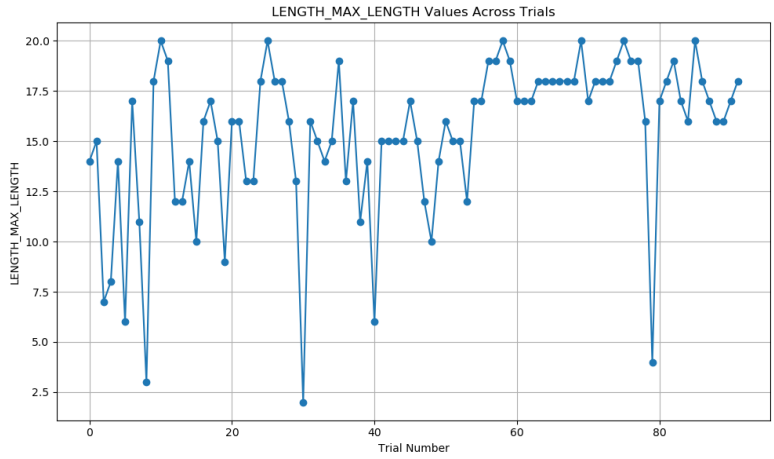

Figure 12: Evolution of LENGTH_MAX_LENGTH during Optuna optimization.

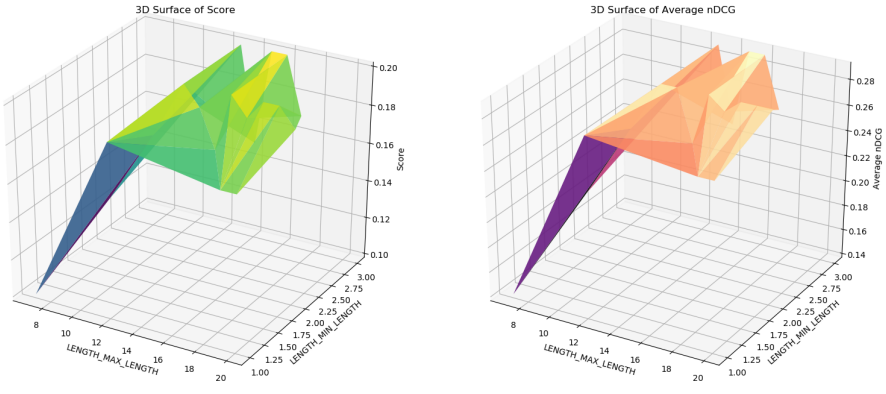

Figure 13: 3D surface visualization of the score landscape across different values of LENGTH_MIN_LENGTH and LENGTH_MAX_LENGTH. Lighter regions correspond to higher scores.

4.7. Score Threshold

In information retrieval systems, the SCORE_THRESHOLD parameter defines the minimum score a document must achieve to be considered relevant and included in the final result set. This threshold acts as a filter to exclude documents with low relevance scores, thereby improving the precision of the

retrieval system. A threshold set too low may include irrelevant documents, reducing precision, while a threshold set too high may exclude relevant documents, reducing recall.

Optimization with Optuna Using Optuna for hyperparameter optimization, we explored various values for the SCORE_THRESHOLD parameter. The optimization process aimed to find a threshold that makes the nDCG better.

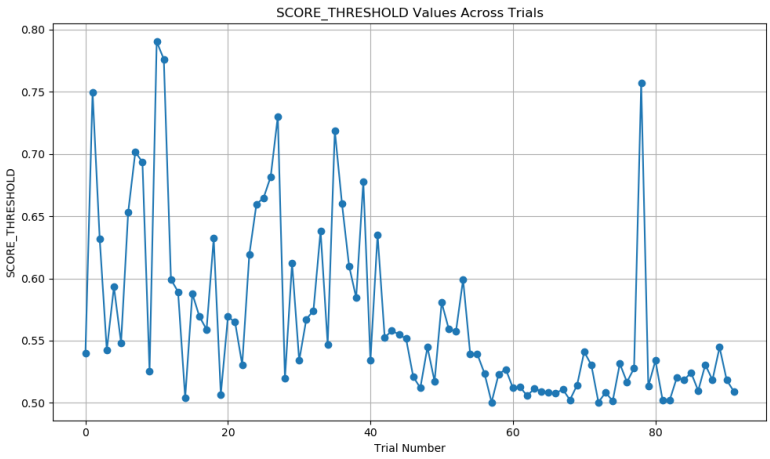

Figure 14: Evolution of SCORE_THRESHOLD during Optuna optimization.

As shown in Figure 14, Optuna identified that setting the SCORE_THRESHOLD to approximately 51% of the highest document score yielded optimal performance. This means that only documents scoring at least half as high as the top-scoring document are considered relevant [23].

4.8. Spellchecking

Our spellchecking mechanism works by first searching the token in the inverted index. If the token is not found, the system attempts to find a similar term within the inverted index and substitutes it in the query.

Two main parameters can be configured:

- SPELLCHECKER_NUMBER_OF_SUGGESTIONS → the number of spell correction suggestions to consider.
- SPELLCHECKER_MIN_TERM_LENGTH → the minimum length of a word to be eligible for spellchecking.

Using Optuna for hyperparameter tuning, we observed that the optimal value for SPELLCHECKER_MIN_TERM_LENGTH is around 4 or 5, while the best value for SPELLCHECKER_NUMBER_OF_SUGGESTIONS is typically 1 or 2. These trends are clearly visible in the figure 15:

4.9. Proximity Reranking

The reranking mechanism works by taking the documents retrieved by the initial search, analyzing the query, and extracting all words pairs from the query. These pairs are then searched within the retrieved documents, considering only occurrences where the two words appear within a configurable maximum distance.

The final document score is computed by combining the BM25 score with the proximity score using configurable weights. As seen in the previous section, the following parameters can be tuned:

- PROXIMITY_RERANKER_MAX_SLOP → the maximum allowed distance between two words for proximity reranking.

- BM25_WEIGHT → the weight assigned to the BM25 score in the final ranking.
- PROXIMITY_WEIGHT → the weight assigned to the proximity score in the final ranking.

In figure 16 we can see that Optuna identified the best value for PROXIMITY_RERANKER_MAX_SLOP as 4, which represents a good trade-off between flexibility and relevance in matching word pairs. Additionally, the optimal weights for combining the scores were approximately 0.6 for BM25 and 0.4 for proximity reranking.

4.10. More infos

To view the details of our TREC Eval runs, you can visit the following link: https://bitbucket.org/upd-dei-stud-prj/seupd2425-basette/src/master/eval/. We have uploaded all of our TREC Eval runs across various parameters, providing a comprehensive overview of the results.

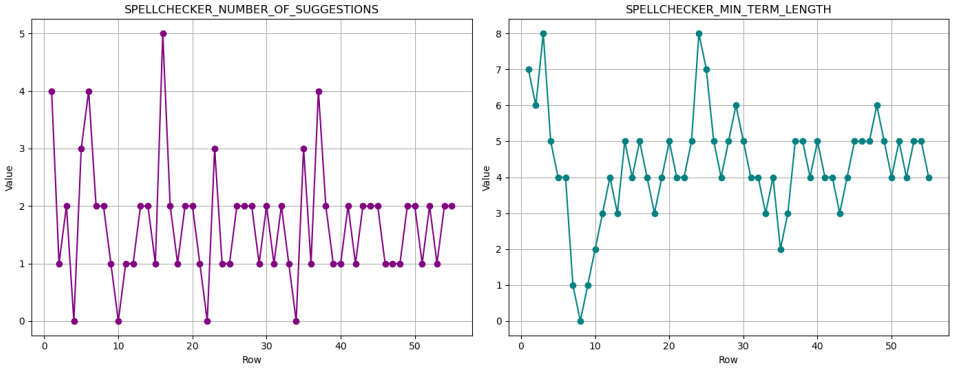

Figure 15: Optuna optimization results for spellchecking parameters.

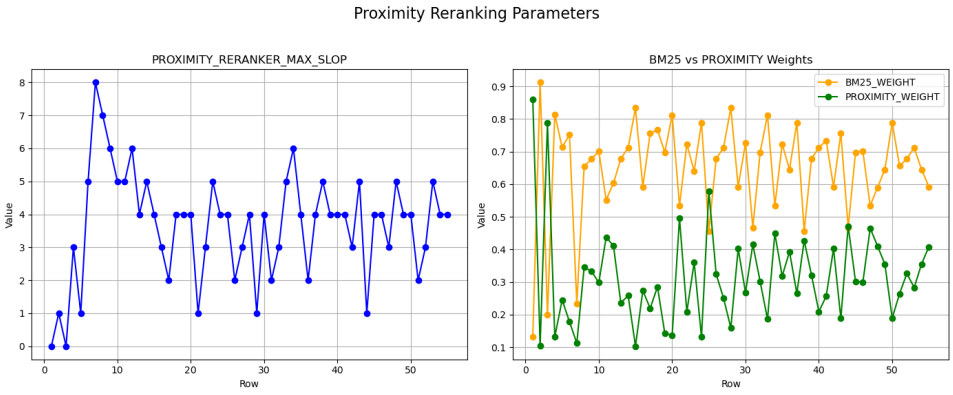

Figure 16: Optuna optimization results for proximity reranking parameters.

5. Results

In the first part of this section, we provide an overview of the retrieval systems we developed, along with their performance on the test set. This allows us to assess and compare their effectiveness under standardized conditions. The evaluation is aimed at understanding how well these systems generalize to unseen data and so how good they are in real-world scenarios.

The assessment was conducted on the snapshots in table 5, which are part of the test set provided by the Longeval benchmark.

Table 5
List of test snapshots used for evaluation.

2023_03-fr	2023_04-fr	2023_05-fr	2023_06-fr	2023_07-fr	2023_08-fr

5.1. Systems

Below is a brief description of the four retrieval systems we analyzed, along with their performance based on the TREC evaluation metrics. Each system represents a different stage of development or enhancement, allowing us to assess the impact of various techniques on retrieval quality.

Table 6
Average NDCG per system (rows) and snapshot (columns).

	2023_03-fr	2023_04-fr	2023_05-fr	2023_06-fr	2023_07-fr	2023_08-fr
default	0.3017	0.2955	0.2988	0.3196	0.2902	0.2404
fine_tuned	0.4158	0.4274	0.4236	0.4471	0.4098	0.3397
spellchecker	0.4219	0.4347	0.4306	0.4538	0.4164	0.3453
reranking	0.4288	0.4417	0.4376	0.4604	0.4231	0.3512

Default This system uses Lucene's default configuration, without any fine-tuning. It serves as our baseline, helping us understand how much improvement has been achieved through our customizations. The system relies on the standard FrenchAnalyzer, includes basic stopword removal, and uses Lucene's default BM25 parameters for the search component. As shown in Table 6, the NDCG scores for this system are consistently lower than the others. This is expected, given that no optimization has been applied, making it a useful reference point for evaluating the effectiveness of our modifications.

Fine-Tuned This version incorporates the parameters obtained through our tuning process using Optuna, as detailed in the training section 4. Unlike the default system, it benefits from targeted adjustments to BM25 and other indexing/search parameters. The results show a clear improvement in NDCG compared to the baseline, confirming that our optimization process has had a positive impact on retrieval performance.

Spell Checking In this system, we added a spell-checking component on top of the fine-tuned configuration. The idea was to test whether correcting potential typos in user queries could lead to better results. Although we were initially unsure of the effect this would have, the system ended up performing slightly better than the fine-tuned version on average across the snapshots. This suggests that spell checking contributes positively in this context, at least with our chosen setup and data.

Re-Ranking The final system extends the spell-checking setup by adding a positional re-ranking step. After retrieving the top-ranked documents, this additional phase re-orders them based on positional scoring. Among all the systems, this one achieved the best overall performance. However, this improvement comes with a cost: query execution time is roughly twice as long compared to the other

systems. Despite the slower response, the gain in relevance indicates that re-ranking is a worthwhile step.

5.2. Statistical Analysis

To understand whether the differences in NDCG scores across the various retrieval systems are statistically meaningful, we carried out a set of analyses of variance (ANOVA).

5.2.1. One-way ANOVA: System Effect

We began with a one-way ANOVA, a statistical test used to determine whether there are significant differences between the means of three or more independent groups, in our case, the four retrieval systems. This type of analysis helps assess whether the choice of system has a real impact on the retrieval performance (measured by NDCG), or whether the observed differences could be due to random chance.

Table 7 reports the ANOVA results. The p-value associated with the `system` factor is extremely small ($p < 0.001$), indicating that the type of system used has a statistically significant effect on the NDCG scores. In other words, not all systems perform the same, and the differences we observe are unlikely to be due to noise.

Table 7
One-way ANOVA results: System effect on NDCG.

	sum_sq	df	F	Pr(>F)
C(system)	679.08	3	1931.91	0.0
Residual	26390.28	225231	–	–

Figure 17 shows the distribution of NDCG scores for each system. It's immediately apparent that the three systems we developed, `fine_tuned`, `spellchecker`, and `reranking`, achieve higher and more consistent scores compared to the `default` system. Among our methods, the `reranking` system performs the best on average, though all three are relatively close to one another.

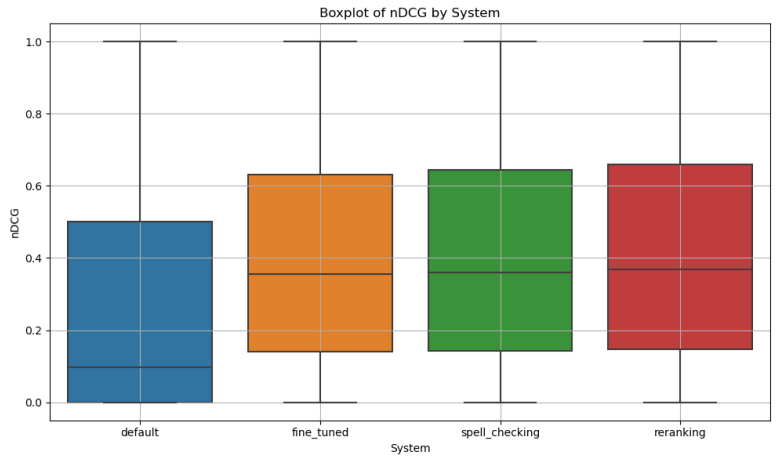

Figure 17: Boxplot of NDCG scores by system.

To explore these differences in more detail, we conducted a Tukey's HSD (Honestly Significant Difference) test. This post-hoc analysis is used after ANOVA to perform pairwise comparisons between all groups and determine exactly which ones differ from each other significantly.

The result is shown in Figure 18, which presents the confidence intervals for all system comparisons. This test confirms what the boxplot already suggests: the `default` system is statistically distinct from the other three, which form a relatively homogeneous group. In fact, Tukey's test effectively identifies

two statistically significant "clusters": one composed of the `default` system, and another grouping together `fine_tuned`, `spellchecker`, and `reranking`. This separation reinforces the idea that the optimizations we introduced led to real, consistent improvements in retrieval performance compared to the baseline.

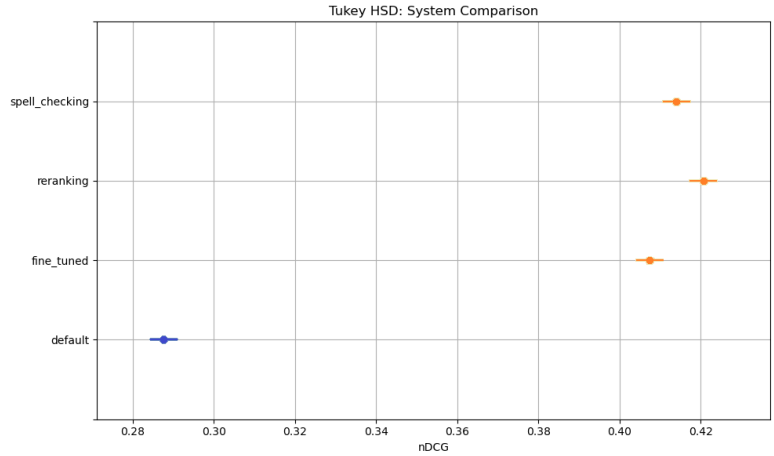

Figure 18: Tukey HSD test for pairwise system comparisons.

5.2.2. Two-way ANOVA: System and Snapshot Interaction

To better understand how both the retrieval system and the specific temporal snapshot influence performance, and whether there is any interaction between the two, we performed a two-way ANOVA. This type of analysis allows us to assess the *independent effects* of each factor (in this case, `system` and `snapshot`), as well as how their *combination* might affect the outcome (NDCG scores).

The summary of the analysis is shown in the table below:

Table 8
Two-way ANOVA results: System and Snapshot effects on NDCG.

	sum_sq	df	F	Pr(>F)
C(system)	679.04	3	1950.47	0.0
C(snapshot)	253.54	5	436.96	0.0
Residual	26136.74	225226	–	–

The results confirm that both factors, `system` and `snapshot`, have a significant effect on NDCG scores ($p < 0.001$ in both cases). This means not only does the choice of retrieval system influence the performance, but so does the specific snapshot of data used for evaluation.

The boxplot in Figure 19 provides a visual summary of these effects. Similar to the one-way case, we observe that the three optimized systems (`fine_tuned`, `spellchecker`, and `reranking`) perform noticeably better than the `default` system. However, a key detail emerges here: performance on the `2023_08` snapshot is consistently lower across all systems. The other snapshots show relatively stable and balanced results, but `2023_08` appears to be an outlier in terms of performance drop.

To dive deeper into these interactions, we ran a post-hoc Tukey HSD test, which identifies specific pairs of group combinations (system $\times$ snapshot) that differ significantly from one another. The results, shown in Figure 20, reveal several statistically distinct groupings.

Notably, the Tukey test separates the combinations into four broad groups:

- The worst-performing group: `default` on `2023_08`.
- A second lower tier: the other systems also on `2023_08`.
- A third group: `default` across all other snapshots.

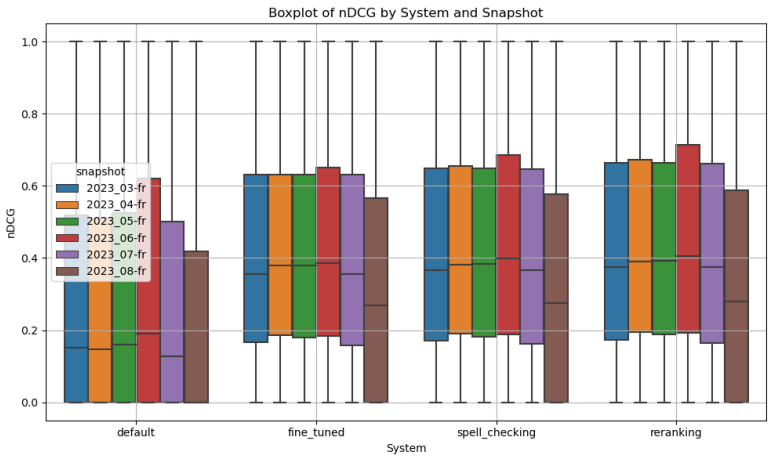

Figure 19: Boxplot of NDCG scores by system and snapshot.

- The top group: all other systems (`fine_tuned`, `spellchecker`, `reranking`) across snapshots `2023_03` to `2023_07`.

This suggests that `2023_08` has a negative effect on system performance in general, regardless of which system is used. It is likely that this drop is not entirely due to the systems themselves, but rather to some property of the `2023_08` snapshot, possibly related to data quality, domain shift, or changes in content distribution during that period.

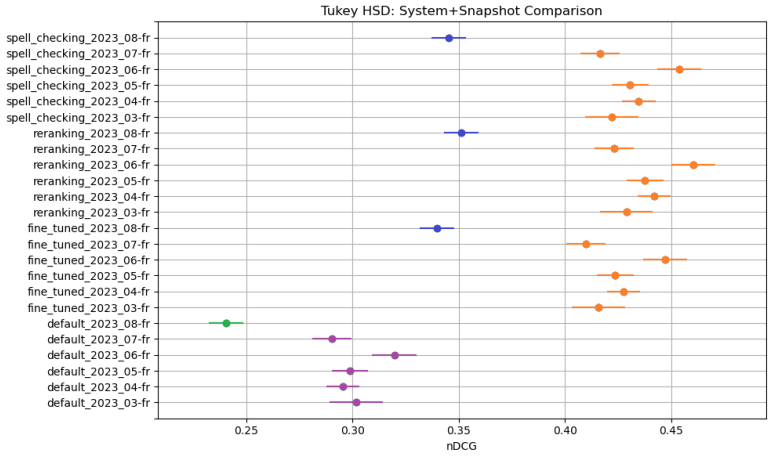

Figure 20: Tukey HSD test for pairwise comparisons (two-way ANOVA).

6. Conclusions and Future Work

In this project, we developed a configurable and multithreaded Information Retrieval system tailored for efficient operation on basic and commonly available hardware, in line with the CLEF LongEval lab's goals. Our guiding principle was to prioritize performance and adaptability without relying on specialized computational resources such as GPUs. To this end, we built a Java-based architecture powered by Lucene, with extensive support for configuration-driven customization of preprocessing, indexing, and retrieval strategies.

We conducted a systematic exploration of classical IR techniques, including text normalization, token filtering, stemming, and stopword removal, as well as the use of n-gram matching and proximity-based reranking. Our system achieved significant performance gains through multithreaded indexing and

querying, with indexing time reduced by nearly an order of magnitude. Additionally, we employed Optuna for automatic hyperparameter optimization, allowing us to fine-tune our system for both retrieval effectiveness and stability over time.

Several advanced techniques, such as semantic expansion via WordNet, temporal alignment using Duckling, and neural reranking with CamemBERT, were considered but ultimately discarded due to either limited performance improvements or prohibitive computational costs. These decisions were consistent with our core objective: building a fast, tunable IR system that works reliably on modest hardware.

Looking forward, we plan to explore lightweight semantic models, expand our support for multilingual corpora, and revisit some of the discarded approaches under improved hardware conditions. Our long-term vision remains consistent with the title of this project: designing an efficient and adaptable IR system for all hardware profiles, capable of balancing retrieval effectiveness with practical performance constraints.

Acknowledgments

The authors thank the organisers of CLEF 2025 LongEval for providing the data and evaluation infrastructure [1].

Declaration on Generative AI

The authors employed ChatGPT (OpenAI) exclusively to (1) refine English grammar and style and (2) rephrase or clarify selected sentences for improved conceptual clarity. All AI-suggested text was thoroughly reviewed, edited, or discarded at the authors' discretion, and no AI tool was used to generate original scientific content, devise research ideas, or draw conclusions. The authors take full responsibility for the accuracy and integrity of the manuscript. This usage is fully compliant with the CEUR-WS Policy on AI-Assisting Tools.

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] Apache, 2023, Lucene v9.5.0, URL: https://lucene.apache.org/core/9_5_0/index.html.

[3] D. Jurafsky, J. H. Martin, N-gram language models, 3rd ed., draft ed., Stanford University, 2025. URL: https://web.stanford.edu/~jurafsky/slp3/3.pdf.

[4] S. Hariri, L. Kurgan, Parallel Computing and Information Retrieval, Springer, 2004.

[5] S. Robertson, H. Zaragoza, The probabilistic relevance framework: Bm25 and beyond, Foundations and Trends in Information Retrieval 3 (2009) 333–389. doi:10.1561/1500000019.

[6] Apache Software Foundation, Apache lucene - similarity models, 2025. URL: https://lucene.apache.org/core/9_9_2/core/org/apache/lucene/search/similarities/BM25Similarity.html.

[7] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, 2019, pp. 2623–2631. URL: https://optuna.org/. doi:10.1145/3292500.3330701.

[8] G. A. Miller, Wordnet: A lexical database for english, Communications of the ACM 38 (1995) 39–41.

[9] M. Didion, extjwnl: Extended java wordnet library, https://github.com/extjwnl/extjwnl, 2014. Accessed: 2025-05-22.
[10] B. Sagot, The Lefff, a freely available and large-coverage morphological and syntactic lexicon for French, in: Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10), European Language Resources Association (ELRA), Valletta, Malta, 2010. URL: http://alpage.inria.fr/~sagot/wolf-en.html, also includes WOLF: WordNet Libre du Français.
[11] O. Lutz, Synonymes - github repository, https://github.com/olup/synonymes, 2021. Accessed: 2025-05-22.
[12] Facebook AI Research, Duckling: A haskell library for parsing temporal expressions, https://github.com/facebook/duckling, 2016.
[13] Facebook, Duckling docker image, https://hub.docker.com/r/facebook/duckling, 2025.
[14] L. Martin, B. Muller, P. J. O. Suárez, Y. Dupont, L. Romary, É. V. de la Clergerie, D. Seddah, B. Sagot, Camembert: a tasty french language model, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, 2020, pp. 7203–7219. URL: https://aclanthology.org/2020.acl-main.645.
[15] A. AI, Deep java library (djl), https://djl.ai, 2025. Accessed: 2025-05-21.
[16] H. Face, camembert-base - hugging face, https://huggingface.co/camembert-base, 2020. Accessed: 2025-05-21.
[17] FasterXML, 2025, Jackson (GitHub Repository), URL: https://github.com/FasterXML/jackson.
[18] C. L. Organizers, 2025, CLEF LongEval Data, URL: https://clef-longeval.github.io/.
[19] S. I. Project, French stopword lists, 2025. URL: https://github.com/stopwords-iso/stopwords-fr/tree/master/raw, accessed: 2025-05-25.
[20] A. Lucene, Frenchanalyzer (lucene 5.0.0 api), 2015. URL: https://lucene.apache.org/core/5_0_0/analyzers-common/org/apache/lucene/analysis/fr/FrenchAnalyzer.html, accessed: 2025-05-25.
[21] D. Turnbull, Practical bm25 - part 3: Considerations for picking b and k1 in elasticsearch, 2019. URL: https://www.elastic.co/blog/practical-bm25-part-3-considerations-for-picking-b-and-k1-in-elasticsearch, accessed: 2025-05-25.
[22] Elastic, Length token filter, 2025. URL: https://www.elastic.co/docs/reference/text-analysis/analysis-length-tokenfilter, accessed: 2025-05-25.
[23] M. Lee, Better rag retrieval — similarity with threshold, 2023. URL: https://meisinlee.medium.com/better-rag-retrieval-similarity-with-threshold-a6dbb535ef9e, accessed: 2025-05-25.