For HW6 add a line describing your approach

SEUPD@CLEF: Team FADERIC.

Enrico Bolzonello[1], Christian Marchiori[1], Daniele Moschetta[1], Riccardo Trevisiol[1] and Fabio Zanini[1]

[1]*University of Padua, Italy*

Abstract

This report analyzes and explains the system developed by Team FADERIC for the LongEval Lab at CLEF 2023, Task 1 - LongEval-Retrieval. The team members are all students of the University of Padua. The system developed is a search engine that has to retrieve documents form a corpus, composed by files in french language, or automatically translated in English.

Keywords

CLEF, LongEval, Information retrieval, Search engines, Retrieve documents,

1. Introduction

Search engines have become an indispensable tool for people to retrieve various kinds of information in their daily routine. However, recent research has shown that *Information Retrieval (IR)* systems tend to perform poorly over time as the test data becomes more distant from the training data. This issue is particularly critical in the field of computer science, where data is constantly updated and information quickly becomes obsolete. Therefore, *Conference and Labs of the Evaluation Forum (CLEF)* 2023 LongEval task has gained interest in evaluating the temporal persistence of IR systems. The aim of this report is to present our solution to this challenge. The paper is organized as follows: Section 2 describes our approach; Section 3 explains our experimental setup; Section 4 discusses our main findings; finally, Section 5 draws some conclusions and outlooks for future work.

"Search Engines", course at the master degree in "Computer Engineering", Department of Information Engineering, and at the master degree in "Data Science", Department of Mathematics "Tullio Levi-Civita", University of Padua, Italy. Academic Year 2022/2023

enrico.bolzonello@studenti.unipd.it (E. Bolzonello); christian.marchiori@studenti.unipd.it (C. Marchiori); daniele.moschetta@studenti.unipd.it (D. Moschetta); riccardo.trevisiol.1@studenti.unipd.it (R. Trevisiol); fabio.zanini@studenti.unipd.it (F. Zanini)

2. Methodology

In this section, we describe the methodology we have adopted in order to develop an IR system for the task.

2.1. Parser

We manually inspected the documents provided by CLEF in order to understand their *structure* and be able to extract the body and ID of each document from them, which is shown in Listing 1.

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

Not needed for HW?

Listing 1: Document format

In order to do that, we created a tool called *parser* that has been essential for extracting information from documents in the specified format used by *Text REtrieval Conference (TREC)*. The parser helps us create structured objects that are used for analysis and indexing within the IR system.

Here are the key components we implemented:

- **`ParsedDocument`**: represents a parsed document to be indexed. It has two attributes: `ID` for the unique identifier of the document and `body` for the document's content. This class provides functionalities to set and retrieve documents' attributes.

- **`DocumentParser`**: represents an abstract class providing basic functionalities to iterate over the elements of a `ParsedDocument`, reading and parsing its content.

- **`LongEvalDocumentParser`**: specific `DocumentParser` for the LongEval corpus. It provides an implementation of a parser for the documents in the TREC format. The parser reads a document and returns a `ParsedDocument` that contains the `ID` and the `body` of the document.

We used the `LongEvalDocumentParser` and the `ParsedDocument` in the indexer to represent the content of the documents that are in the directory specified by `docsDir`. The first one has been used to iterate over the content of the specified document, while the second one has been used to represent a document to be indexed.

2.2. Analyzer

In order to *process texts* from documents and queries, we have implemented custom analyzers: since the collections are provided in both French and English language, two of them have been implemented.

2.2.1. French analyzer

The `FrenchLongEvalAnalyzer` component is in charge of processing French language texts, it is composed by:

- **Tokenizing**: the `StandardTokenizer` is used, which exploits the Word Break rules from Unicode Text Annex #29 [1];
- **Character folding**: the `ICUFoldingFilter` is used, which applies the foldings from Unicode Technical Report #30 [2]. This is useful to fold upper cases, accents and other kinds of complex characters;
- **Elision removal**: the `ElisionFilter` is used, which removes the elision from words;
- **Stopword removal**: the `StopFilter` is used, which removes the given stopwords from the tokens. In this system we have tried using the default Lucene [3] stoplist and custom one, generated by picking the 50 most frequent terms in the documents;
- **Position filtering**: a custom `TokenFilter` has been implemented to set the `positionIncrement` attribute of all tokens to a specific value. This will be useful to ignore the `positionIncrement` due to removed stopwords in the search phase when we will use the proximity between tokens, as explained in Subsection 2.4.3;
- **Stemming**: this process is useful to reduce words to their base form, in this system we have tried using the Snowball French [4] and Light [5] stemmers.

In Table 1 we show an example of the analyzing process for the French language.

Table 1
French analyzer process

Step	Tokens
	La méthode d'analyse de texte est essentielle pour l'extraction d'informations.
Tokenizing	[La, méthode, d'analyse, de, texte, est, essentielle, pour, l'extraction, d'informations]
Character folding	[la, methode, d'analyse, de, texte, est, essentielle, pour, l'extraction, d'informations]
Stopword removal (50 most freq.)	[methode, analyse, texte, essentielle, extraction, informations]
Stemming (Ligth)	[method, analys, text, esentiel, extraction, inform]

2.2.2. English analyzer

The `EnglishLongEvalAnalyzer` component is in charge of processing English language texts, it is composed by:

- **Tokenizing**: the `StandardTokenizer` is used, which exploits the Word Break rules from Unicode Text Annex #29 [1];
- **Character folding**: the `ICUFoldingFilter` is used, which applies the foldings from Unicode Technical Report #30 [2]. This is useful to fold upper cases, accents and other kinds of complex characters;
- **Possessive removal**: the `EnglishPossessiveFilter` is used, which removes the very frequent possessives ('s) from words;
- **Stopword removal**: the `StopFilter` is used, which removes the given stopwords from the tokens. In this system we have tried using the default Lucene [3] stoplist and custom one, generated by picking the 50 most frequent terms in the documents;
- **Position filtering**: a custom `TokenFilter` has been implemented to set the `positionIncrement` attribute of all tokens to a specific value. This will be useful to ignore the `positionIncrement` due to removed stopwords in the search phase when we will use the proximity between tokens, as explained in Subsection 2.4.3;
- **Stemming**: this process is useful to reduce words to their base form, in this system we have tried using the Snowball English (Porter2) [6] and the Krovetz [7] stemmers.

In Table 2 we show an example of the analyzing process for the English language.

Table 2
English analyzer process

Step	Tokens
	The text analysis method's importance lies in its role in information extraction.
Tokenizing	[the, text, analysis, method, importance, lies, in, its, role, in, information, extraction]
Character folding	[the, text, analysis, method, importance, lies, in, its, role, in, information, extraction]
Stopword removal (50 most freq.)	[text, analysis, method, importance, lies, role, information, extraction]
Stemming (Krovetz)	[text, analysis, method, importance, lie, role, information, extraction]

2.3. Indexer

Indexing is a crucial step where we create a searchable database, called *index*, for the parsed documents. The index contains important information about the documents, such as the words and phrases they contain, their frequency, and their location within the document. Indexing allows us to store the documents in a *structured* manner, which greatly speeds up the retrieval process by enabling users to search for documents based on keywords or phrases. To achieve this, we developed the following components:

- **`DirectoryIndexer`**: indexes the documents located in a specified directory and its sub-directories. It accepts various parameters, including the directory containing the documents to be indexed, the `DocumentParser`, the `Analyzer`, the `Similarity` to be used for indexing, the expected number of documents and the location where the index will be stored. Our code ensures that the document directory is readable and the index directory is writable before initiating the indexing process. Additionally, it keeps track of statistics, such as the number of indexed files and documents.
  The main component of the class is the `index` method, which is in charge of performing the actual indexing of the documents. This method iterates through the documents in the directory, extracting their content and adding it to the index. During all the iterations, some statistics indexing is given, such as the time taken every 10 thousand documents. Finally, the index is closed.

- **`BodyField`**: represents the body field of a document. This field has the following properties:
  - it is *tokenized*, meaning that the body is broken into words, or tokens, to make the search more accurate and flexible;
  - *frequencies* and also the *positions* of the tokens are stored, in order to allow for phrase queries with proximity, as explained in Subsection 2.4.3;
  - the body content is *stored*, even if this had an impact on the index size, this was needed in the search phase in order to pass documents bodies to the reranker, as explained in Subsection 2.4.6.

In Table 3 are reported the index performances obtained using the analyzers described in Subsection 2.2 and the setup described in Section 3.

Table 3
Indexing performances

Collection	**Docs size** (GB)	**Stoplist**	**Stemmer**	**Body terms**	**Index size** (GB)	**Time** (s)
French	7.99	Default	Snowball	7,497,875	6.98	1224
		50 most freq.	Light	7,459,058	6.95	842
English	7.27	Default	Snowball	7,253,947	6.49	1041
		50 most freq.	Krovetz	7,451,647	6.43	848

Shingle is Lucene Jargon. Better Word N-gram

2.4. Searcher

In the searcher we have used a *boolean query* approach, in this way it was possible to create complex queries by combining, using the *boolean operators*, different components to be matched. The component we have used in our queries are explained in Subsections from 2.4.1 to 2.4.6.

2.4.1. BM25

Since the document collection has a significant size, as a base of our system we opted for a classic BM25[8] approach due to the *efficiency* of it and higher *effectiveness* compared to other methods.
The Lucene [3] implementation has default values $k1 = 1.2$ and $b = 0.75$ which worked fine, but we decided to fine-*tune* the parameters to improve measures, in particular, *Normalized Discounted Cumulated Gain (nDCG)* since it is the relevant measure for the LongEval campaign. The results of our experiments are reported in Table 4.

Run		FADERIC_French-Stop50-Stem-Shingle-Fuzzy							
Measure		ndcg							
		k1							
		0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
b	0,30	0,3941	0,3952	0,3954	0,3957	0,3963	0,3936	0,3948	0,3934
	0,40	0,3967	0,398	0,3984	0,3992	0,3992	0,3992	0,3981	0,3975
	0,50	0,3999	0,4004	0,4008	0,4013	0,4014	0,4014	0,4014	0,4011
	0,60	0,3999	0,4013	0,4017	0,4025	0,4026	0,4024	0,4019	0,4008
	0,70	0,4021	0,4029	0,4034	0,4038	0,4043	**0,4047**	0,4046	0,4039
	0,75	0,4018	0,4028	0,4035	0,4039	0,4038	0,4043	0,4037	0,4035
	0,80	0,401	0,4025	0,4033	0,404	0,4041	0,4043	**0,4047**	0,4039
	0,90	0,3985	0,3998	0,4009	0,4014	0,4018	0,4021	0,4015	0,4011

Table 4
NDCG results with different BM25 parameters

The best result was given by two pairs: $k1 = 1.8, b = 0.8$ and $k1 = 1.6, b = 0.7$; we chose pair $k1 = 1.8, b = 0.8$ due to having the highest *Mean Average Precision (MAP)*. Note that the performance gain was minimal, just $0,006$ from the score with default parameters, which was expected.

2.4.2. Fuzzy

A *fuzzy* search, or approximate search, is a technique used to search for approximate or *partial* matches between a search term and documents in a collection of texts. Unlike exact search, in which the match must be exact and precise, fuzzy search allows you to find results even when the words you search for do not exactly match those in the documents. Fuzzy search is particularly useful when you want to get results even if there are *misspellings*, *language variants*, *abbreviations* or other forms of variations in the search terms or texts of the documents. For edit

example, if you search for the term "roam" with a fuzzy search, the document containing the term "foam" might also be returned.

What changes?

Fuzzy search techniques are based on the use of algorithms that evaluate the similarity between text strings. One of the most common algorithms used for fuzzy search is the Levenshtein algorithm [9], which calculates the ~~editing~~ distance between two strings, that is, the minimum number of operations (insertions, deletions, and character substitutions) required to transform one string into the other. Lucene [3], for example, uses a variant of the algorithm just described, the Damerau-Levenshtein algorithm, named after the Damerau algorithm [10].

Lucene [3] also allows you to add an additional (optional) parameter with which to specify the maximum number of changes allowed. In our case we decided to set a manual *threshold* to choose the value of the parameter; if the word length is greater than or equal to the threshold then the fuzzy parameter is set to 2 otherwise 1 is used. The threshold is called "fuzzyThreshold" and can be set in the configuration file; we decided to set it to 10. Finally, to avoid performance degradation, in our IR system, fuzzy search is applied only if the query contains a single term.

2.4.3. Shingles

Word N-grams

Shingles are a sentence analysis technique of dividing the words of a sentence into sequences of n consecutive words. For example, for the sentence "the dog barks," we can create the n-grams "the dog" and "dog barks." Shingles are useful because they capture *local relationships* between words within a text, so this approach helps identify similar, though not identical, phrases and can *improve search relevance*. The maximum number of words within a shingle can be set in the configuration file in `maxShingleSize`; in our case, we decided to generate shingles with a maximum of 3 words. Also, we avoided generating unigrams, i.e., shingles with exactly one word, as they do not capture any local relationships within the query.

We then decided to set up a proximity search within each shingle. The proximity of terms in a shingle can be used to identify documents in which the search terms occur in a certain *spatial relationship*. For example, if we are searching for the terms "dog" and "brown" with the proximity of 3 words, we want to find documents in which these two terms appear within a maximum distance of three words from each other. Thus, if a document contains sentences such as "I saw a brown dog in the park" or "The brown dog was running fast," these documents would be considered relevant because the terms "dog" and "brown" are close to each other. In our case, the proximity parameter is set to 5.

Finally, we applied a boost to all shingles based on the size of the shingle itself.

2.4.4. SpellChecker

Did you find spell errors in the query?

In the searcher, we tried to implement a spell checker for the query tokens. For every query token, using the `SpellChecker` class of Lucene [3], the spell checker analyzes the token and, based on a dictionary of words stored in the resources folder, suggests one similar (not by meaning but by *spelling*) word. This new word is appended at the end of the query, after the original token, and is given a weight of 0,2 to the new word. This is done because if we match the original token, it's more relevant than matching the new modified token. We do this only for those queries that are composed of only one word (token), because, otherwise, the

If no, why use a spell checker?

performance dropped. In fact, for these queries, suggesting one similar but different word can make a difference in case the one word inserted in the query is *misspelled.* In the case of queries composed of more than one token, it's difficult that all the words are misspelled, so by changing all the words, the performances obviously drop.

2.4.5. Synonyms

Managing synonyms in a retrieval system is not straightforward. The use of synonyms may not necessarily improve results, and this depends on how they are used.

Synonyms may be useful in the context of IR to broaden the search to include more related terms. However, the introduction of synonyms can also create problems such as noise in the information retrieval process. Here are some reasons why the use of synonyms may not improve results:

- **Polysemy**: words may have *more than one meaning*. Introducing synonyms might lead to an increase in irrelevant results if a synonym is associated with another meaning of the searched word. For example, if one searches for "bank" in the context of a financial bank, introducing the synonym "riverbank" could generate irrelevant results.
- **Irrelevant synonyms**: not all synonyms are equally *relevant in the context* of a given query. Some synonyms might be too general or too specific concerning the user's intent, leading to inappropriate or insufficient results.
- **Redundancy**: adding synonyms can lead to redundancy in the answers provided. If multiple synonymous words or phrases are used in the query, there may be significant overlap between the results, reducing the overall effectiveness of the IR system.

However, it should be kept in mind that the effectiveness of using synonyms in improving the results of the IR system also depends on the specific implementation and the characteristics of the domain or context in which the system is used. In some cases, the use or *expansion* of synonyms could actually improve the accuracy of information retrieval.

Furthermore, the use of synonyms can increase computational complexity in the information retrieval process. Because synonyms require accurate correspondence with indexed documents, additional computations must be performed to identify and compare matching synonyms in indexed texts.

We made several attempts to implement synonyms in our system; a summary description of what we did is given below.

Firstly, since handling synonyms in the index takes too much computation time, we decided to handle them directly in the search. We added synonyms in the queries with a query expansion approach.

In addition, we decided to use the WordNet dictionary[11]. WordNet is a lexical database that groups English words into sets of synonyms called synsets, providing semantic relationships and definitions. It offers a comprehensive resource for natural language processing tasks, such as word sense disambiguation, information retrieval, and sentiment analysis. Since WordNet is written in C, it was necessary to use an additional API in order to use the dictionary on our system. That API is called extJWNL (Extended Java WordNet Library)[12] and does not need the WordNet database installed locally.

Also, to improve dictionary lookup, we decided to use the OpenNLP [13] library for natural language processing and limit polysemy. Each word in the original query was processed by an OpenNLP *Part of Speech (PoS)* Tagger in order to obtain the tag associated with the word. That function analyzes the context in which the word is used and returns the associated tag. The model used for the pos tagger was `en-pos-maxent.bin` and the tags are associated with WordNet section as shown in Table 5.
Knowing the tag of each word in a query made it possible to look up the word in the corresponding dictionary section. For example, for the query "free software," free was searched in the adjective section and software in the noun section. This strategy improved the metrics very little probably because the queries provided by Long Eval are very short, averaging 2/3 words. In addition, OpenNLP works well with *properly formulated sentences*, including consideration of capitalization and punctuation. In this case, queries are very crudely formulated, for example, some begin with a capital letter and some do not, as a result, OpenNLP does not always provide the correct tag. Then, this strategy might be more useful in the case of more complex queries, such as those characterizing a conversational IR system.

Table 5
OpenNLP Tags compared with WordNet Sections

OpenNLP Tag	WordNet Section
JJ	Adjectives
VB	Verbs
RB	Adverbs
NN	Nouns
Others	No synonyms retrieved

Subsequently, we tried to give a *different boost* to each synonym. As a first approach, we decided to provide a boost based on the amount of synonyms returned. In this case, the boost was calculated in this way:

$$boost = \frac{BoostBase}{SynonymListLength} \tag{1}$$

This approach was used to limit the importance associated with each synonym if the returned synonym list is long, being more likely to get *irrelevant synonyms*.

Finally, we moved synonym management, creating two new Analyzers: `SynonymAnalyzer` and `SynonymPOSAnalyzer`, which are applied only in the search part:

- **`SynonymAnalyzer`**: uses as input a query already previously analyzed with the standard Analyzer, i.e. `EnglishLongEvalAnalyzer` or `FrenchLongEvalAnalyzer`, after applies: `SynonymGraphFilter`, `FlattenGraphFilter` and `StopFilter`.

SynonymGraphFilter represents a filter that can be directly applied to a TokenStream within an Analyzer. The filter creates a synonym graph based on specified configurations and expands the terms found in the analyzed text by adding the corresponding synonyms to the token graph. FlattenGraphFilter converts an incoming graph token stream, such as one from SynonymGraphFilter, into a flat form so that all nodes form a single linear chain with no side paths. Every path through the graph touches every node. This is necessary when indexing a graph token stream because the index does not save PositionLengthAttribute and so it cannot preserve the graph structure. However, at search time, query parsers can correctly handle the graph and this token filter should not be used.
This Analyzer uses a list of synonyms in .txt format, available in two versions: standard and custom. Before being processed by the Analyzer, the synonym list is transformed into a SynonymMap object via the AnalyzerUtil's loadSynonymList function.

- **SynonymPOSAnalyzer**: takes as input EnglishLongEvalAnalyzer, then applies an OpenNLPPOSFilter, so that each word is assigned the associated tag. Then, it applies a custom filter called SynonymPOSFilter to manage the tags and look up words in the WordNet dictionary. Creating a custom filter was not trivial as the information about it in the documentation and online is very limited. The filter allows us to:
  1. fetch the input tokens,
  2. retrieve their associated tag,
  3. search the synonyms in the dictionary based on their tag,
  4. process the synonyms with the standard analyzer i.e. EnglishLongEvalAnalyzer
  5. and return as output a TokenStream containing all the synonyms found.

The tokenStream returned as output by both Analyzers is then transformed into a list of strings which is in turn processed by the Searcher to apply a boost. Finally, the synonyms are added to the BooleanQuery along with the original query.

satisfactory

2.4.6. Reranker

After all of the previous steps, we obtained a working system that achieved decent results at a good speed, so we tried to improve it by introducing a second stage, called *passage re-ranking*, in which each of the documents returned by the first retrieval model would be scored and re-ranked by a more computationally-intensive method involving Machine Learning.
We achieved this by leveraging a library called PyGaggle[14], which provides some deep neural architectures for text ranking and question answering, and two transformer-based models, T5 and BERT, with different checkpoints and even our own trained checkpoint using code from another library[15].

— space

In our system, we can choose how many documents are to be reranked from the top for two reasons: the first is raw computing performance, reranking all 1000 retrieved documents takes a long time and we don't have machines powerful enough to handle it; the second is that we saw that reranking more than 50 documents ~~drops~~ our measures ~~down~~, even below the baseline

lowers

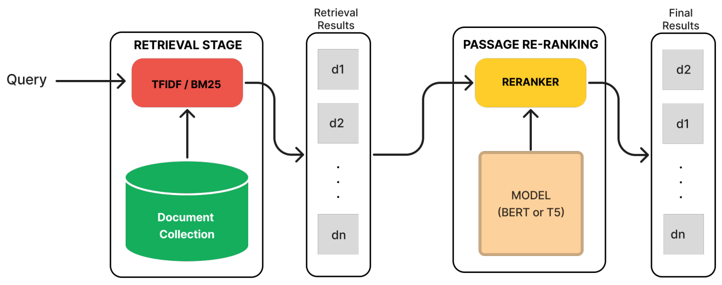

Figure 1: Retrieve-than-rerank framework

measure without reranking.
Our first approach to reranking was to consider only the scores returned by the models to rerank the documents but we then switched to an approach where we consider also the BM25 score as follows. Let $Score_{BM25}(i)$ the score given by BM25 for the document at rank i and $Score_{rr}(i)$ the score given by the reranker for the document at rank i, and let n be the total number of reranked documents, we define:

$$nScore_{rr}(i) = \left(Score_{rr}(i) + \min_{j\in[1,n]} Score(j)\right) \cdot \frac{Score_{BM25}(1)}{Score_{rr}(1)} \tag{2}$$

as the normalized score for the reranked documents, since the models returned a score in the range $[-10, +10]$, which was not suitable for our case.
In our first approach, we simply passed the score to Lucene's `ScoreDoc` object and we were done. But in this way, we would lose information about the ranking given by BM25, which is still relevant, so we defined a new score:

$$finalScore(i) = mntr + (1-\alpha) \cdot Score_{BM25}(i) + \alpha \cdot nScore_{rr}(i) \tag{3}$$

where $mntr$ is the maximum score from docs which are not reranked, in this way, we preserve the order of this docs. With this approach, we can give a weight to the reranker to find the balance and we do not lose the information given to us by the first stage. Note that $\alpha = 1$ corresponds to considering only scores from the reranker.

Pretrained Models

We tried two transformer-based models, T5 and BERT since they are supported by PyGaggle. First, we tried T5[16], but with BERT[17] we got better results. Starting from the same base model, t5-base[18] for T5 bert-base-uncased[19] for BERT, we tried different checkpoints[1] fine-tuned specifically for reranking tasks and we even tried to train our checkpoint. The pre-trained checkpoints that we used are:

[1]saving the model's parameters and optimizer state during the training process

space before \cite

- *monot5-base-msmarco-10k*[20]
- *bert-base-mdoc-bm25*[21]

The results for the T5 model are reported in Table 6 and the results for the BERT model are reported in Table 7.

Table 6
monot5-base-msmarco-10k model with different number of documents to rerank

	nDCG	**map**
0	0,4075	0,2411
10	**0,414**	**0,2502**
20	0,4119	0,2477
50	0,4083	0,242
100	0,405	0,2376
250	0,3987	0,2301

Table 7
bert-base-mdoc-bm25 model with different number of documents to rerank

	nDCG	**map**
0	0,4075	0,2411
10	0,4207	0,2608
20	**0,4222**	**0,2617**
50	0,4212	0,2598
100	0,4184	0,2563
250	0,4104	0,2478

The BERT pretrained checkpoint with 20 reranked documents improved nDCG by 3.56% and MAP by 8.5%.

Training our own checkpoint

At this point, BERT gave us good results so we took it a step further and we tried to find ways to finetune it to our data. The training process is pretty straightforward:

- **Data pre-processing**. Transformers expect batches of tensors as input, so we need to preprocess our data to the expected format. For processing textual data the tokenizer tool is used, which splits text into tokens; in our case, we exploit the pre-trained BERT tokenizer which returns tokens that are not necessarily words, but rather subwords: frequently used words are (or should) not split into smaller subwords, but rare words should be decomposed into meaningful subwords [22]. Further, the tokenizer adds, at the beginning and at the end, two special tokens, respectively `[CLS]` and `[SEP]`. The tokenizer returns a dictionary with three items:
  - `input_ids`, indices corresponding to each token in the sentence
  - `attention_mask`, indicates whether a token should be attended to or not

  - `token_type_ids`, identifies which sequence a token belongs to when there is more than one sequence

  An important note is that BERT accepts input sequences of up to 512 tokens, so the tokenizer truncates longer documents.
- **Training**. The easiest way to train is to use the Trainer API from PyTorch [23].

To ease development, we used a package for training deep language model rerankers[15] and adapted the example code to our collection. The trainer in the package expects ~~the~~ training data in a JSON file with the format shown in Listing 2.

```
{
    "qry": {
        "qid": str,
        "query": List[int],
    },
    "pos": List[
        {
            "pid": str,
            "passage": List[int],
        }
    ],
    "neg": List[
        {
            "pid": str,
            "passage": List[int]
        }
    ]
}
```

not needed for HW1

Listing 2: Train format

The `convert_to_training.py` takes care of converting data to the training file, given the ranking file, the qrels, the query collection and the docs collection. Then it is sufficient to use the `trainer.py` code to get the trained model. Unfortunately, we don't have access to sufficiently powerful machines so we were forced to train on a Google Colab notebook. This came with a major drawback: the maximum runtime is 12 hours, so we couldn't train on more than 1 epoch since a 2 epoch model was estimated to take 15 hours to train. This obviously tanked our model performances, and, as we can see in Table 8.
The Colab notebook with all the hyperparameters can be found at https://colab.research.google.com/drive/1oFeYSkR31A-MUibwWrfNcKLUyPxEYbOq?usp=sharing.
The trained model can be found at https://huggingface.co/enricobolzonello/clef_longeval.

Integrating with Lucene

One problem that emerged while working with the reranker was integrating it with Lucene since the reranker is written in Python and our main program is in Java. To solve the issue we came up with three approaches:

Table 8
Our trained model with different number of documents to rerank

	nDCG	**map**
0	0,4075	0,2411
10	0,3910	0,2253
20	0,3741	0,1975
50	0,3405	0,1580

1. passing intermediate text files, with documents, query to the reranker and returning the ranking to Java. This solution was used for initial testing but was deemed too inefficient and prone to errors
2. using Python as the system's entry point and using the PyJNIus library[24] to access Java classes. The same approach was used by Birch [25], but we should have changed the classes too much to integrate tightly the reranker and more importantly we didn't want to change the entry point of our system
3. our final solution was to call in some way Python from Java

The library we used to achieve this is JEP [26], which uses *Java Native Interface (JNI)* and the CPython API to start up the Python interpreter inside the *Java Virtual Machine (JVM)*. Thanks to the Python interpreter, we can call, at each query, the reranker which returns a list of generic `Objects` that is converted to a list of `Float`.

3. Experimental Setup

The experimental setup of this project consists of the following:

- The *project* is available at https://bitbucket.org/upd-dei-stud-prj/seupd2223-faderic/src/master/;
- The *collections* are taken from https://clef-longeval.github.io/data/;
- The *evaluation tool* used is trec_eval v9.0.7, available https://trec.nist.gov/trec_eval/;
- To compute the runs we have used the following *hardware*: CPU AMD Ryzen 7 2700, GPU Zotac GeForce RTX 2060 6 GB, RAM 16 GB DDR4;

In order to reproduce the runs for this system, it is necessary to follow the instructions given in the project's ReadMe file.

4. Results and Discussion

We have *combined* the different components explained in Section 2 and we conducted a thorough experimentation process to identify the best-performing system for our task. By tuning the parameters of each component and selecting the optimal ones, we were able to build a system that addresses the persistence issue of IR systems. In Table 9, Figure 2 and Figure 3 are reported the MAP and nDCG scores obtained on those systems.

The keywords reported in the names of the runs have the following meanings:

- French: used documents and queries from the French collection
- English: used documents and queries from the English collection
- BM25: used Okapi BM25 similarity (with default parameters)
- BM25Tuned: used Okapi BM25 similarity, with parameters tuned on training collection: k1=1.6 and b=0.7
- LMDirichlet: used Dirichlet smoothing (with default parameter)
- StopDefault: used Lucene's default stop words list
- Stop50: used stoplist built by picking the 50 most frequent terms in the documents indexed without stoplist and stemming
- LightStem: used the Light stemmer
- KStem: used Krovetz stemmer
- SnowStem: the Snowball stemmer
- Shingle: used word shingles (max window size = 3) query expansion
- Fuzzy: used fuzzyness (threshold parameter = 10) query expansion
- SynCustom: used custom synonyms list
- SynPOS: used WordNet synonym list together with OpenNLP PoS tagging
- SpellCheck: used the spell checker
- ReRerank20W6: used reranker, reranking 20 documents with weight 0.6 given to the reranker scores
- ReRerank30: used reranker, reranking 30 documents with weight 1 given to the reranker scores
- TrainedRerank30: used reranker, reranking 30 documents using our custom model

Table 9
MAP and nDCG values table

sort by descending map

Run name	**MAP value**	**nDCG value**
FADERIC_French-BM25-StopDefault-SnowStem	0.211	0.3786
FADERIC_French-LMDirichlet-Stop50-LightStem	0.1731	0.3398
FADERIC_French-BM25Tuned-Stop50-LightStem-Shingle-Fuzzy	0.2383	0.4047
FADERIC_French-BM25-Stop50-LightStem-Shingle-SpellCheck-SynCustom	0.2404	0.4066
FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom	0.2416	0.4079
FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom-Rerank20W6	0.2671	0.4274
FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-Rerank30	0.2632	0.423
FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-TrainedRerank30	0.1799	0.3599
FADERIC_English-BM25-StopDefault-SnowStem	0.149	0.2927
FADERIC_English-LMDirichlet-Stop50-KStem	0.1228	0.2612
FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-SynPOS	0.1634	0.3081
FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-Rerank30	0.1873	0.3527
FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-SynPOS-Rerank30	0.1877	0.3271

what is the additional info wrt Table 9? You could [illegible]

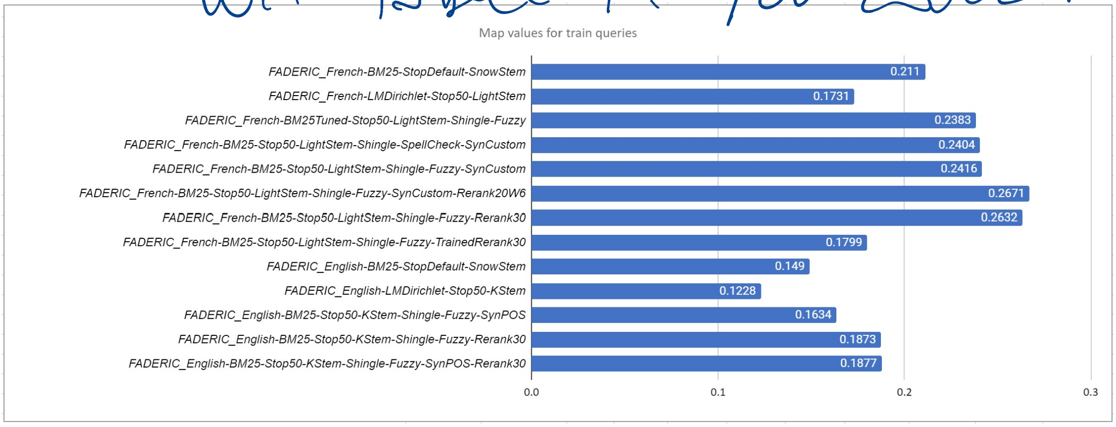

Figure 2: MAP values table

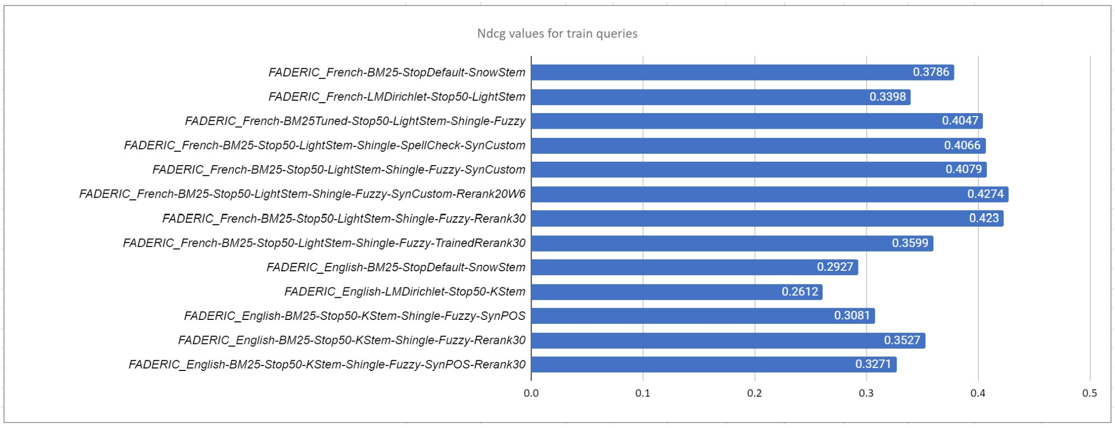

Figure 3: nDCG values table

During the tuning of our system, we primarily focused on MAP and nDCG. However, in order to perform a more *comprehensive* analysis, we also considered additional measures such as Precision and Recall: the first one is the fraction of retrieved documents that are relevant to the user's query, indicating a measure of the accuracy of the system, while with the second is a measure of the completeness of the system in retrieving all relevant results, computed by the fraction of relevant documents retrieved.

In Figure 4, we show the *interpolated* Precision-Recall curve, which can be useful to show the inverse relationship between Precision and Recall, indicating the *trade-off* between these two measures.

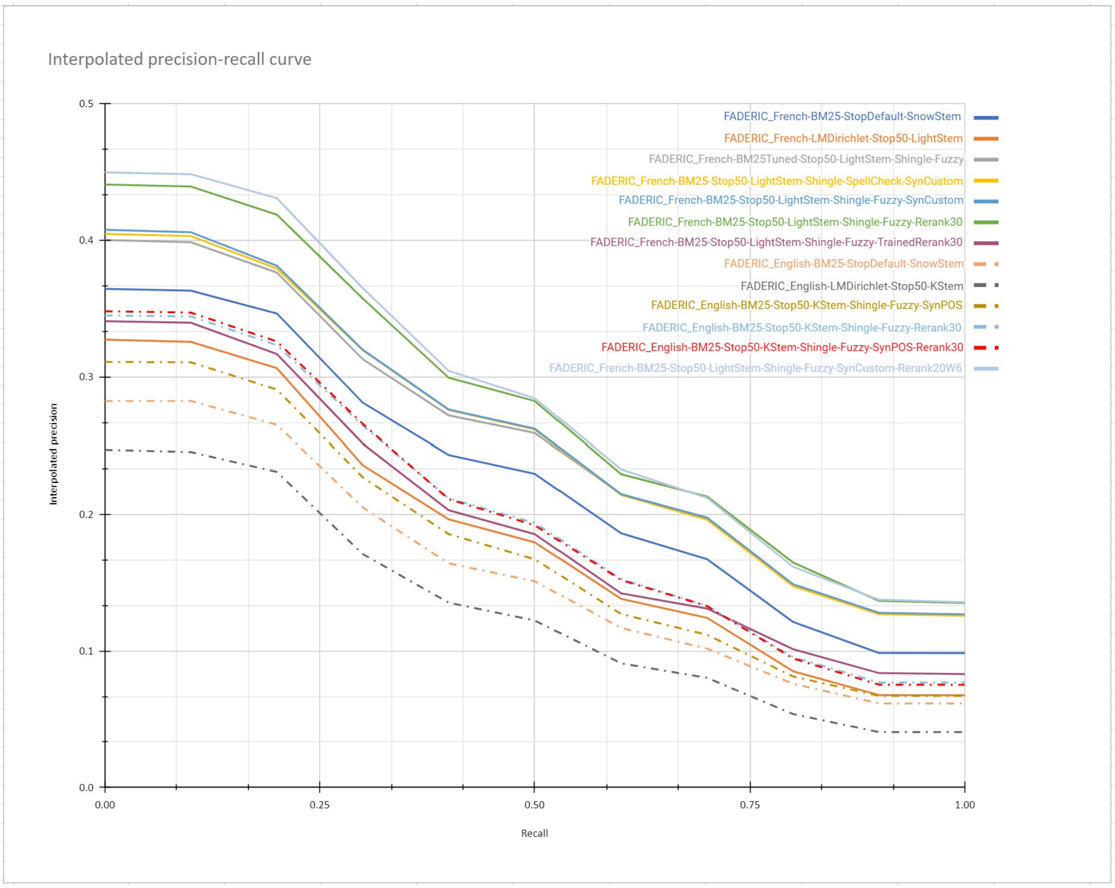

Figure 4: Interpolated Precision-Recall curve

Label

In Table 10 and in Table 11 we have reported a more complete list of scores respectively for the French and English runs. For space reasons, we target the runs as:

- fr_1 = FADERIC_French-BM25-StopDefault-SnowStem
- fr_2 = FADERIC_French-LMDirichlet-Stop50-LightStem
- fr_3 = FADERIC_French-BM25Tuned-Stop50-LightStem-Shingle-Fuzzy
- fr_4 = FADERIC_French-BM25-Stop50-LightStem-Shingle-SpellCheck-SynCustom
- fr_5 = FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom
- fr_6 = FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom-Rerank20W6
- fr_7 = FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-Rerank30
- fr_8 = FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-TrainedRerank30
- en_1 = FADERIC_English-BM25-StopDefault-SnowStem
- en_2 = FADERIC_English-LMDirichlet-Stop50-KStem
- en_3 = FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-SynPOS
- en_4 = FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-Rerank30
- en_5 = FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-SynPOS-Rerank30

Table 10
Measures for French runs

runid	all	fr_1	fr_2	fr_3	fr_4	fr_5	fr_6	fr_7	fr_8
num_q	all	672	672	672	672	672	672	672	672
num_ret	all	658471	658512	660838	665307	660838	660838	660838	660838
num_rel	all	2626	2626	2626	2626	2626	2626	2626	2626
num_rel_ret	all	2271	2156	2323	2311	2316	2316	2318	2318
map	all	0.211	0.1731	0.2383	0.2404	0.2416	0.267	0.2632	0.1799
gm_map	all	0.0545	0.0394	0.0696	0.0696	0.0702	0.0786	0.0765	0.0552
Rprec	all	0.1747	0.1469	0.1946	0.1975	0.1987	0.228	0.2278	0.1365
bpref	all	0.3753	0.3561	0.4063	0.4079	0.4085	0.4128	0.4122	0.3701
recip_rank	all	0.3424	0.3134	0.3734	0.3787	0.3824	0.4222	0.414	0.317
iprec_at_recall_0.00	all	0.3646	0.3274	0.4003	0.4048	0.4078	0.4499	0.441	0.3409
iprec_at_recall_0.10	all	0.3633	0.3258	0.3987	0.4032	0.406	0.4485	0.4395	0.3398
iprec_at_recall_0.20	all	0.3465	0.3066	0.3765	0.3795	0.3815	0.431	0.4189	0.3169
iprec_at_recall_0.30	all	0.2813	0.2359	0.3131	0.3197	0.32	0.3651	0.3575	0.2515
iprec_at_recall_0.40	all	0.2433	0.1963	0.272	0.276	0.2763	0.3046	0.2995	0.203
iprec_at_recall_0.50	all	0.2296	0.1795	0.2592	0.2618	0.2623	0.2846	0.2825	0.1855
iprec_at_recall_0.60	all	0.1861	0.1381	0.2147	0.2141	0.2147	0.2328	0.2293	0.1421
iprec_at_recall_0.70	all	0.1671	0.1242	0.1974	0.196	0.1976	0.212	0.213	0.131
iprec_at_recall_0.80	all	0.1212	0.0851	0.1481	0.1474	0.1489	0.1616	0.1647	0.1013
iprec_at_recall_0.90	all	0.0985	0.0677	0.1276	0.1269	0.1279	0.1375	0.1366	0.0838
iprec_at_recall_1.00	all	0.0984	0.0675	0.1267	0.1258	0.1268	0.1357	0.1351	0.0831
P_5	all	0.1637	0.1375	0.1827	0.1866	0.1875	0.2074	0.2065	0.1315
P_10	all	0.1332	0.1088	0.1469	0.1463	0.1472	0.1603	0.156	0.1007
P_15	all	0.109	0.0889	0.1167	0.1176	0.1183	0.1233	0.1243	0.0869
P_20	all	0.0901	0.0753	0.099	0.0986	0.099	0.099	0.1022	0.0799
P_30	all	0.0683	0.058	0.0743	0.0746	0.0748	0.0748	0.0743	0.0742
P_100	all	0.0267	0.0232	0.028	0.0279	0.0279	0.0279	0.0279	0.0279
P_200	all	0.0146	0.0133	0.0151	0.015	0.015	0.015	0.015	0.015
P_500	all	0.0064	0.006	0.0065	0.0065	0.0065	0.0065	0.0065	0.0065
P_1000	all	0.0034	0.0032	0.0035	0.0034	0.0034	0.0034	0.0034	0.0034
recall_5	all	0.2106	0.1802	0.2417	0.2477	0.2478	0.2738	0.2732	0.1749
recall_10	all	0.3402	0.2789	0.3828	0.3813	0.383	0.4167	0.4042	0.2606
recall_15	all	0.4203	0.338	0.4499	0.454	0.456	0.4724	0.4757	0.3366
recall_20	all	0.4595	0.3814	0.5051	0.5016	0.5032	0.5034	0.5193	0.4119
recall_30	all	0.5187	0.4412	0.5677	0.5703	0.571	0.571	0.5657	0.5651
recall_100	all	0.6688	0.5816	0.7028	0.7019	0.7022	0.7022	0.7019	0.7019
recall_200	all	0.7283	0.6729	0.7587	0.7553	0.7555	0.7555	0.756	0.756
recall_500	all	0.7994	0.7574	0.8235	0.8217	0.8219	0.8219	0.8232	0.8232
recall_1000	all	0.8485	0.8072	0.8685	0.8656	0.8663	0.8663	0.8662	0.8662
ndcg	all	0.3786	0.3398	0.4047	0.4066	0.4079	0.4274	0.423	0.3599

Table 11
Measures for English runs

runid	all	en_1	en_2	en_3	en_4	en_5
num_q	all	671	671	672	672	672
num_ret	all	654919	654764	655602	655602	655602
num_rel	all	2623	2623	2626	2626	2626
num_rel_ret	all	1878	1765	1924	1915	1924
map	all	0.149	0.1228	0.1634	0.1873	0.1877
gm_map	all	0.0164	0.0122	0.0195	0.0215	0.0221
Rprec	all	0.1253	0.1092	0.1385	0.1708	0.1706
bpref	all	0.3263	0.3139	0.3417	0.3524	0.3536
recip_rank	all	0.2697	0.2364	0.2967	0.3289	0.3322
iprec_at_recall_0.00	all	0.2825	0.2471	0.3111	0.3451	0.3482
iprec_at_recall_0.10	all	0.2825	0.2455	0.3108	0.3444	0.3472
iprec_at_recall_0.20	all	0.2652	0.231	0.2909	0.3234	0.3261
iprec_at_recall_0.30	all	0.205	0.1709	0.227	0.2646	0.2658
iprec_at_recall_0.40	all	0.1641	0.1353	0.1855	0.2118	0.2111
iprec_at_recall_0.50	all	0.151	0.1223	0.1671	0.1937	0.192
iprec_at_recall_0.60	all	0.1168	0.0909	0.127	0.1526	0.1519
iprec_at_recall_0.70	all	0.1017	0.0803	0.1118	0.1331	0.1328
iprec_at_recall_0.80	all	0.076	0.0538	0.0813	0.0955	0.0945
iprec_at_recall_0.90	all	0.0616	0.0406	0.0671	0.0769	0.0753
iprec_at_recall_1.00	all	0.0616	0.0406	0.0671	0.0769	0.0752
P_5	all	0.1195	0.1019	0.1345	0.1542	0.1542
P_10	all	0.0954	0.0793	0.1042	0.1131	0.1137
P_15	all	0.0784	0.065	0.0835	0.0898	0.0904
P_20	all	0.0662	0.0543	0.0705	0.0732	0.0745
P_30	all	0.0513	0.0433	0.0537	0.053	0.0536
P_100	all	0.0201	0.0179	0.0208	0.0209	0.0208
P_200	all	0.0113	0.0103	0.0116	0.0115	0.0116
P_500	all	0.0051	0.0048	0.0052	0.0052	0.0052
P_1000	all	0.0028	0.0026	0.0029	0.0028	0.0029
recall_5	all	0.1512	0.1329	0.171	0.2017	0.202
recall_10	all	0.2437	0.1988	0.2628	0.2887	0.2909
recall_15	all	0.297	0.2432	0.3155	0.3404	0.343
recall_20	all	0.3318	0.2727	0.3523	0.3683	0.3745
recall_30	all	0.3861	0.3281	0.4018	0.397	0.401
recall_100	all	0.5	0.4486	0.5151	0.5167	0.514
recall_200	all	0.563	0.518	0.5774	0.5754	0.5769
recall_500	all	0.6443	0.6087	0.6612	0.6597	0.6607
recall_1000	all	0.7027	0.6635	0.7186	0.715	0.7186
ndcg	all	0.2927	0.2612	0.3081	0.3257	0.3271

Based on these results, we can derive the following considerations:

- the runs on the French collections are significantly better than the ones on the English one, the reason for this is the *automatic translation* of the document collection, which has led to errors and inconsistencies in the English ~~one; which has resulted in errors and inconsistencies in the English~~ collection.
- the system performs better when configured to use BM25 similarity instead of the Dirichlet smoothing;
- the *tuned* parameters on BM25 just gave slightly betters results, probably *overfitting* the run we have tuned them on, for this reason we have chosen to use the default ones in most of the runs;
- the custom stoplist we have generated by picking the most frequent terms has outperformed Lucene's default ones because since it was based on the specific collection the *effectivness* of using a stoplist has been maximized;
- in both French and English the Snowball stemmer performed worse than the Light and Krovetz stemmer, respectevly;
- fuzzyness and spell checking have given similar results but the first one was slightly better, probably because the spell checker is dictionary based and thus *not flexible* for non conventional words;
- the use of shingles improved the performances, allowing to have more *contextual matches* by looking for group of words instead of single ones;
- synonyms have slightly improved the performances, this is due to the fact that sometimes they can be *misleading* and retrieve documents that are not contextual with the query;
- the reranking is a very *powerful* tool that has given a huge performance increase to our runs.

Based on the previous results and these considerations, we have decided to submit to CLEF the following systems:

- FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom-Rerank20W6, i.e., the best system overall;
- FADERIC_English-BM25-Stop50-KStem-Shingle-Fuzzy-SynPOS-Rerank30, i.e., the best system overall on the English collection;
- FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-SynCustom, i.e., the best system without the use of reranking;
- FADERIC_French-BM25-Stop50-LightStem-Shingle-Fuzzy-Rerank30, i.e., the best system without the use of synonyms;
- FADERIC_French-BM25Tuned-Stop50-LightStem-Shingle-Fuzzy, i.e., the best system without the use of both synonyms and reranking;

5. Conclusions and Future Work

From the obtained results we understand that *query expansion* and *reranking* play a major role in the IR systems. Those two features granted us the biggest performance improvements. Although the system has reached decent scores, our work could be *further developed* in different ways.

With respect to query expansion, we could improve the synonyms feature by using other dictionaries, since WordNet is quite outdated and a big portion of the queries were regarding very recent topics, and also by using better French dictionaries, since there are only a few available when working with languages other than English and we had to customize our own. Another query expansion option could be switching from simple dictionaries to *neural models*, in order to expand a query with words related not just to the meaning of the single words but also to the *context* of the whole topic the user is looking for.

Regarding the reranker, there are some things we could improve, mainly focusing on the *trained model.* As discussed in Section 2.4.6, we did not have the required hardware to properly train a machine learning model, so we trained on Colab with a time limit of 8 hours. This forced us to train with only 1 epoch and we tested only one set of hyperparameters, so, assuming having the necessary hardware, future improvements could focus on training a better model, with more epochs and testing different hyperparameters.
More affordable improvements could be done by dropping the libraries for training and inference and working directly with Torch for the interaction with the model, so we could be able to use the latest Torch version, which includes a new implementation of the Transformer API which speeds up training significantly.

References

[1] C. Chapman, 2022, Unicode Text Segmentation, URL: https://www.unicode.org/reports/tr29/.

[2] A. Freytag, 2004, Character Foldings, URL: https://www.unicode.org/reports/tr30/tr30-4.html.

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

[4] M. Porter, R. Boulton, F. Brault, 2002, Snowball French stemmer, URL: https://snowballstem.org/algorithms/french/stemmer.html.

[5] J. Savoy, Light stemming approaches for the french, portuguese, german and hungarian languages, in: Proceedings of the 2006 ACM Symposium on Applied Computing, Association for Computing Machinery, 2006, pp. 1031–1035.

[6] M. Porter, R. Boulton, 2002, Snowball English stemmer, URL: https://snowballstem.org/algorithms/english/stemmer.html.

[7] R. Krovetz, Viewing morphology as an inference process, Artificial Intelligence 118 (2000) 277–294.

[8] S. E. Robertson, U. Zaragoza, The Probabilistic Relevance Framework: BM25 and Beyond, Foundations and Trends in Information Retrieval (FnTIR) 3 (2009) 333–389.

[9] V. I. Levenshtein, et al., Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet physics doklady, volume 10, Soviet Union, 1966, pp. 707–710.

[10] F. J. Damerau, A technique for computer detection and correction of spelling errors, Communications of the ACM 7 (1964) 171–176.

[11] Princeton University, 2010, WordNet, URL: https://wordnet.princeton.edu/.

[12] Extended Java WordNet Library (extJWNL), 2011. URL: https://github.com/extjwnl/extjwnl.

[13] Apache, 2023, OpenNLP, URL: https://opennlp.apache.org/.

[14] PyGaggle, 2021. URL: https://github.com/castorini/pygaggle.

[15] L. Gao, Z. Dai, J. Callan, Rethink training of bert rerankers in multi-stage retrieval pipeline, in: The 43rd European Conference On Information Retrieval (ECIR), 2021.

[16] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, Journal of Machine Learning Research 21 (2020).

[17] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[18] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, 2020, t5-base, URL: https://huggingface.co/t5-base.

[19] Google, 2018, bert-base-uncased, URL: https://huggingface.co/bert-base-uncased.

[20] castorini, 2021, monot5-base-msmarco-10k, URL: https://huggingface.co/castorini/monot5-base-msmarco-10k.

[21] L. Gao, 2021, bert-base-mdoc-bm25, URL: https://huggingface.co/Luyu/bert-base-mdoc-bm25.

[22] HuggingFace, 2023, Subword tokenization, URL: https://huggingface.co/docs/transformers/tokenizer_summary#subword-tokenization.

[23] HuggingFace, 2023, Trainer, URL: https://huggingface.co/docs/transformers/main_classes/trainer.

[24] PyJNIus, 2022. URL: https://github.com/kivy/pyjnius.
[25] Z. A. Yilmaz, S. Wang, W. Yang, H. Zhang, J. Lin, Applying BERT to Document Retrieval with Birch, Technical Report, University of Waterloo, 2019.
[26] Jep, 2022. URL: https://github.com/ninia/jep.