



INFORMATION RETRIVAL

ADRIAN MARTINEZ FUENTES, UO295454



15 DE DICIEMBRE DE 2024
UNIVERSITY OF OVIEDO
ESCUELA DE INGENIERIA INFORMATICA

Contenido

Introduction to the project.....	3
Exercise1 - Indexing	4
Processing the collection.....	4
Tokenisation and creation of the BM25 index	5
Chatbots Prompts	6
Exercise2 – BM25 variants	7
Setup	7
Execution flow.....	8
Program output	9
ANNEX.....	10
Exercise3 – Query expansion using LLR.....	11
Expand process.....	11
Conclusion	12
ANNEX.....	13
Exercise4 – Query expansion metrics.....	14
Execution.....	14
Conclusions.....	14
1. Effect of Parameter n (Number of Initial Documents Retrieved)	14
2. Effect of Parameter m (Number of Terms Added to the Query)	15
Comparison table	15
Exercise5 – Comparison against Semantic Search	16
Setup & Execution	16
Comparative results	16
Comparative Analysis.....	16
Precision (Macro and Micro)	16
Recall (Macro and Micro).....	17
F1 Score (Macro and Micro).....	17
Reflection on Results	17
Lucene	17
Semantic Retrieval.....	17
Individual queries analysis	18
Final Conclusion	18
ANNEX.....	19

OptionalEx – Hybrid Search with Reranking	20
Setup & Execution	20
Changes made to the notebook	20
Lexical Search Implementation (BM25):.....	20
Semantic Search Implementation:	20
Query Submission Functions:	21
Hybrid Search Approach:	21
Main Execution Flow:	21
Reranking Methods Tested	21
Original Search Results	21
Reranking Step	21
Impact on the quality of the result	22
Table of values:	22
Summary and Conclusion	22
Set of screenshots of the program	23
Conclusions of the project	23

Introduction to the project

Before starting with the implementation of each of the exercises I want to clarify the environment, the modules and the collection that I have used for this deliverable:

- 1) As a programming environment I have used Visual Studio Code together with a virtual environment and python scripts
- 2) The collection that has been used for this project is the NF-Corpus one.
- 3) For the achievements of each of the exercises, several imports have been made in each script, not only from external libraries, such as bm25s, chromaDb, etc. but also from previous exercises to reuse the code and avoid duplication, making our scripts more efficient and manoeuvrable.

Also, the structure of the scripts is the following:

- 1) At the top we have the imports and the global variables (if needed)
- 2) The script is divided into a set of functions, each performing one task
- 3) The invocation to those functions is done from a main function which is executed when running the script.

Exercise1 - Indexing

For this assignment I reviewed the first two notebooks of the laboratory sessions, together with the theoretical concepts explained in class.

In this first script I first imported the following modules into my virtual environment using the pip command from the terminal of vsCode:

- 1) Module `bm25s`: to perform the tokenization, indexing, etc.
- 2) Module `Stemmer`: to create the Stemmer object for `bm25`
- 3) Module `os`: operative system module to create directories
- 4) Module `tarfile`: to compress the indexes into a '.tar.gz' files
- 5) Module `re`: to work with regular expressions

Processing the collection

The first thing I did inside the parsing function was to open the corpus file and read its content. For each line of the file:

- 1) It was separated by tabulator to obtain the Id and the text for the document and a Json object was constructed. This Json object was added to the verbatim version of the corpus, meanwhile to the plain text version we only added the textual content of each document.
- 2) Then, once we finished the process for all the lines, both lists were returned.

Tokenisation and creation of the BM25 index

After processing the collection, the next step was to create a stemmer (in this case for English). Now, we need to tokenize the plain text version of the corpus we obtained in the previous step.

For this purpose, we will use stop words removal to reduce noise as much as we can and the stemmer object we have just created. In addition, the show progress option is established to give feedback to the user of the tokenization process.

Once the corpus was, we had available the normalized and tokenized version of the text of each document. The next step was to create a BM25 object (contained inside the bm25s module) and pass to it the right parameters in order it to return a retriever object.

These parameters are:

- 1) The corpus (we pass the verbatim version of it)
- 2) The method to be used by BM25 to calculate the scores (in this case Lucene)
- 3) The IDF flavour (Lucene also) to calculate the IDF.

Now that we have the BM25 object initialized and stored inside the 'retriever' variable, we proceed to create the inverted index using the tokenized corpus.

- ➔ Note: during this step, the BM25 is doing several tasks, such as counting the tokens repetition in each document or creating statistics about the document's length, among others.

Finally, we have all set, so the only thing remaining is to save both the index and the structured version of the documents (verbatim one) inside a directory with a descriptive file name. Also, each file is compressed into a 'tar.gz' file in case the person executing this program wants to upload the indexes to an external service.

The compression is done by the usage of the module 'tarfile', which allows given a path and a filename to compress that filename into a 'tar.gz file'.

To finish, also remark that this script has a function for loading an index given its directory and its name using the load function inside the BM25 algorithm.

Furthermore, this script provided error management to avoid showing ugly messages to the user if something fails.

Chatbots Prompts

For the resolution of this exercise, I consulted ChatGPT about the compression of a file, to know about if it was possible to do such task in Python. Here is the conversation:

Prompt: 'I am working on a Python project and I need to be able to compress a file in '.tar.gz' format in a given directory. tar.gz' format in each directory.'

Chatbot response: 'Hello, of course you can. To compress files in the '.tar.gz' format in Python, you can use the tarfile module, which is included by default in the Python standard library. This module allows you to create .tar archives and then compress them using gzip, thus creating '.tar.gz' files.

The process usually involves opening a tar archive in write mode with compression (w:gz), adding the files you want to compress and then closing the compressed archive. This way you can make sure that the files are correctly packed and compressed.'

Exercise2 – BM25 variants

To complete Exercise 2, I used the theoretical slides and the second notebook from the laboratory sessions as reference.

This task involved analysing the performance of different BM25 variants using various text preprocessing configurations.

1. BM25 variants:
 - a. Robertson, Atire, BM25L, BM25+, Lucene
2. Text preprocessing configurations:
 - a. With/without stop word removal, with/without stemming

This setup resulted in 20 experimental configurations. For each configuration, the program retrieved the 100 most relevant documents per query and saved the results to a Json file.

The relevance judgments were extracted from the 'train.3-2-1.qrel' file, and the queries were taken from the 'train.nontopic-titles.queries' file.

These files allowed evaluating the results.

- ➔ Note that, for the relevance judgements, we are only considering those with score greater than 1, as those with score 0 are not present and those with score 1 are marginally relevant for the query.

Setup

We need to import several modules for this exercise. These are:

- 1) bm25s
- 2) Stemmer
- 3) Json module
- 4) The operating system module
- 5) From previous exercise, we reuse the parsing, indexing & loading functions

Then, we initialize several variables:

- 1) The maximum number of documents to be retrieved (100)
- 2) The different bm25 variants
- 3) The different text pre-processing options
- 4) The stemmer object (in English)

Execution flow

The first thing we do inside the main function of this script is the creation of an index for each configuration, that is, 20 indexes. Those indexes are created and saved using the previous script functions. Note that this first function can last time until completed, since we need to create and save 20 different indexes. The aim is to execute this function only once, and then leave it commented in the code for faster execution.

Once we have all the indexes created and saved, we perform the proper bm25 iteration. But first, we need to obtain the queries and the relevance judgements from the different files we have in our collection. For that purpose, we have two functions in charge of that.

Then, what we do is to go through all possible configurations, and for each configuration we retrieve the proper index and iterate all over the queries we have just loaded. For each query, we tokenize it using the corresponding configuration.

Once the query is tokenized, we need to extract both the documents it retrieves and the score for each of the documents. This is done inside the 'retrieve_documents' function.

Then, for each query evaluation we create a list in which we store dictionaries inside it, where each dictionary is composed of two values:

- 1) The Id of each document returned for the query
- 2) The score of that document for that query

Once all the queries for that configuration have been processed, we constructed the descriptive file name based on the configuration used to store the results.

We also obtain the metrics by calling the 'calculate metrics' function, that we saw during the laboratory sessions. This function calculates the micro and macro averaged precision, recall and f1 values together with the precision, recall and f1 for each of the queries. In order this function to perform its task, we need to provide to it the relevance judgements and the 'run' collection, which stores for each query the previous mentioned list.

After calculating all the metrics, we saved the results to a Json file using the previous filename created and we appended both the filename and the metrics to an array that will print by console a summary of the execution when the program finishes.

This report is printed formatted inside the 'print_results_summary' function.

Program output

After the execution of this program, with all the data extracted, we could construct a table in Excel representing for each configuration its micro & macro averaged values:

Model	Macro-Precision	Macro-Recall	Macro-F1	Micro-Precision	Micro-Recall	Micro-F1
robertson-stopwords-stemming	0.0636	0.3046	0.1043	0.0636	0.2368	0.1003
robertson-NONE-stopwords-stemming	0.0620	0.2982	0.1025	0.0620	0.2308	0.0978
robertson-stopwords-NONE-stemming	0.0606	0.2880	0.1001	0.0606	0.2255	0.0955
robertson-NONE-stopwords-NONE-stemming	0.0594	0.2842	0.0983	0.0594	0.2209	0.0936
atire-stopwords-stemming	0.0635	0.3032	0.1043	0.0635	0.2365	0.1002
atire-NONE-stopwords-stemming	0.0621	0.2983	0.1022	0.0621	0.2310	0.0979
atire-stopwords-NONE-stemming	0.0605	0.2880	0.1000	0.0605	0.2252	0.0954
atire-NONE-stopwords-NONE-stemming	0.0594	0.2840	0.0980	0.0594	0.2209	0.0936
bm25l-stopwords-stemming	0.0635	0.3039	0.1039	0.0635	0.2364	0.1001
bm25l-NONE-stopwords-stemming	0.0623	0.2995	0.1024	0.0623	0.2319	0.0982
bm25l-stopwords-NONE-stemming	0.0605	0.2879	0.0999	0.0605	0.2252	0.0954
bm25l-NONE-stopwords-NONE-stemming	0.0595	0.2855	0.0985	0.0595	0.2214	0.0938
bm25+-stopwords-stemming	0.0635	0.3032	0.1043	0.0635	0.2365	0.1002
bm25+-NONE-stopwords-stemming	0.0621	0.2983	0.1022	0.0621	0.2310	0.0979
bm25+-stopwords-NONE-stemming	0.0605	0.2880	0.1000	0.0605	0.2252	0.0954
bm25+-NONE-stopwords-NONE-stemming	0.0594	0.2840	0.0980	0.0594	0.2209	0.0936
lucene-stopwords-stemming	0.0635	0.3032	0.1043	0.0635	0.2365	0.1002
lucene-NONE-stopwords-stemming	0.0621	0.2983	0.1021	0.0621	0.2310	0.0979
lucene-stopwords-NONE-stemming	0.0605	0.2881	0.1000	0.0605	0.2252	0.0954
lucene-NONE-stopwords-NONE-stemming	0.0594	0.2840	0.0980	0.0594	0.2209	0.0936

If we analyse this output, we see that the best f1 macro value is repeated among several Bm25 algorithms, but all of those have in common the stop words removal and the stemming application.

As we can choose what algorithm of the ones tied to choose, we will go with the “lucene” variant, since is the used, for example, by ElasticSearch and OpenSearch. So, our final configuration would be

- 1) Bm25 variant: lucene
- 2) Stop words removal (y/n): yes
- 3) Stemming (y/n): yes

ANNEX

1) This prompt was entered to ChatGPT to generate the 'load_relevance_judgements' function of the second exercise:

'Write a Python function called `load_relevance_judgements` that reads a given file containing 4 columns: `query_id`, `skip`, `corpus_id`, and `score`. It should return a dictionary where the `query_id` maps to another dictionary of `corpus_id` and its relevance score (only keep scores greater than 1). The column named `skip`, that is, the second column of the file, should be ignored. Also, handle invalid lines.'

Chatbot response: it constructed the function shown in the code, with little variations, and an explanation. The function code is not included to not occupy too much space

Explanation:

- **Function Purpose:** `load_relevance_judgements` reads a file containing relevance judgements, filters and stores only relevant information, and returns it in the form of a dictionary.
- **File Format:** The file has four columns: `query_id`, `skip`, `corpus_id`, and `score`. The `skip` column is ignored.
- **Relevance Score:** Only records with a score greater than 1 are kept, indicating relevance.
- **Invalid Lines:** Any lines that do not conform to the expected format (four columns) or contain invalid data (non-integer scores) are skipped.
- **Returned Structure:** The function returns a dictionary where each `query_id` maps to another dictionary that maps `corpus_id` to its respective score.

Additionally, for writing JSON files in the 'save_results' function of the exercise 2, the following prompt was entered (again to CHATGPT):

'Write a Python code snippet that opens a file in write mode with UTF-8 encoding, and writes a Python dictionary into the file in JSON format with indentation'

The code snippet would look like this:

```
python Copiar código

with open(output_file, "w", encoding="utf-8") as f:
    json.dump(result_data, f, indent=4)
```

Explanation:

1. `open(output_file, "w", encoding="utf-8")`:
 - This line opens a file specified by `output_file` in write mode ("w"). The `encoding="utf-8"` ensures that the file is handled using UTF-8 encoding, which is commonly used for text files.
2. `json.dump(result_data, f, indent=4)`:
 - This line writes the `result_data` (which is a Python dictionary) to the file `f`.
 - The `json.dump` function serializes the `result_data` dictionary into a JSON-formatted string and writes it to the file.
 - The `indent=4` argument specifies that the JSON should be pretty-printed with an indentation of 4 spaces, making the output more readable and easier to inspect.

This is a concise and efficient way to store data in JSON format while ensuring it's human-readable.

Exercise3 – Query expansion using LLR

This exercise implements a query expansion system based on the log-likelihood ratio (LLR) test, a technique for identifying ‘surprising’ or relevant terms in a set of documents.

We will decompose the code and logic, aligning them with the steps required in the brief.

General Objective: expand an initial query using the most relevant terms, calculated using the signed LLR and the BM25 index, to improve search results.

Expand process

The first thing which is done inside the query expansion function is the selection of the documents which ID is present in the relevant document IDs returned for the original tokenized query. This is done to filter and work only with relevant documents.

Then, we found for those relevant documents their term frequencies, using the ‘calculate_term_frequencies’ function, passing both the relevant documents and the stemmer as parameters. This function allows to measure which terms are most frequent in the relevant documents, which helps to identify potentially useful terms to expand the query. Once we have the relevant terms for the relevant documents, we sum the frequencies of all terms to get the total number of terms in the relevant documents.

Going back to the expand query function, now that we have the relevant terms for the relevant documents, we need to compute the relevant terms for the rest of the documents in the corpus, stored in the ‘tf_non_relevant’ variable.

For that purpose, we will use the function ‘compute_tf_excluding_relevant’, which uses the global term frequencies calculated before, the relevant term frequencies and the stemmer object. This function is shown below.

This function aims to calculate the term frequencies for the subset of non-relevant documents. It also calculates the total cumulative frequencies for these terms in non-relevant documents.

Then, we invoke the ‘compare_frequencies’ function extracted from the provided Java implementation in the statement, passing the relevant & non-relevant frequencies and establishing a threshold of 0.0 to not take values with negative LLR.

This function is designed to compare the frequency of terms between two sets of documents: relevant and non-relevant. The function calculates a statistical score for each term based on the term frequencies (TF) in both sets and compares these scores against a given threshold.

The function first computes the total term frequencies for both the relevant and non-relevant sets by summing the values in `tf_relevant` and `tf_non_relevant`. This gives the total number of occurrences of all terms in each document set.

Next, the function iterates over the terms in '`tf_relevant`' and compares their frequency in the relevant set with their frequency in the non-relevant set. For each term, the function calculates a score using the **root log-likelihood ratio**.

The score is computed using the formula for the log-likelihood ratio test, which compares the observed frequencies of terms in both sets with the expected frequencies if there were no association. If the calculated score for a term is greater than or equal to the specified threshold, the term is added to a list called '`scored_terms`', along with its score. After all terms have been processed, the '`scored_terms`' list is sorted in descending order based on the score, so that terms with the highest scores appear first.

Conclusion

Finally, once we have the ranked terms, we iterate over them selecting those that are not present in the query. We select as much terms as specified. Finally, we expand the query using the '`join`' function. Then, the query is returned, and the main function tokenizes it and retrieves its documents and scores to print them by the console.

Note that, to prove the correct behaviour of this script, there is a main function which performs query expansion for $n = 5$ (documents retrieved) and $m = 5$ (number of terms we add to the original query). Its results are saved to the '`results-ex3`' directory.

ANNEX

In this annex it is included the prompts entered to the different IAs for this exercise.

1) From the Java implementation of the Signed Root LLR algorithm provided in the statement, a prompt was entered to chat gpt asking for translation to Python. The code of the class is not included because it will take too much space, but the prompt was this one:

‘Having this Java class that performs an execution of the Signed Root LLR algorithm, I need you to translate this code to Python in order to understand it and being able to include the relevant parts of it inside a script that performs query expansion with BM25’

Again, the result of ChatGPT was the whole class translated and include that here will take a lot of space, but note that from that translation I obtained some functions that I use in my script and are enumerated here:

- 1) The `x_log_x` function & the entropy function
- 2) The form to obtain the `k11`, `k12`, `k21` & `k22` values & the LLR
- 3) The function ‘`compare_frequencies`’

2) Also, note that I have used for calculating the term frequencies of the corpus the function you provided us by email (this is not a proper IA prompt, but I considered it to be sufficiently remarkable to be included somewhere in the annex).

3) Also, this prompt was entered to Copilot IA in the visual studio code chat to ask for sorting a collection in Python using a specific criterion:

‘If I have a list of tuples, and I want to sort it according to the second element of each tuple, in descending order, what would be the correct way to do it?’

To sort a collection in Python based on a specific criterion, you can use the `sorted()` function, which allows you to customize the sorting logic using the `key` argument. If you want to sort a list of tuples by the second element of each tuple in descending order, here's how you can do it:

```
python
ranked_terms = [(term1, score1), (term2, score2), (term3, score3)]
sorted_terms = sorted(ranked_terms, key=lambda x: x[1], reverse=True)
```

Here's what's happening in the code:

1. `ranked_terms`: This is your list of tuples, where each tuple contains two elements, for example, a term and its score.
2. `key=lambda x: x[1]`: This tells `sorted` to use the second element of each tuple (`x[1]`) as the sorting key.
3. `reverse=True`: This specifies that the sort should be in descending order.

After running this code, the list `sorted_terms` will contain the tuples sorted from highest to lowest based on the second element of each tuple. For example:

```
python
ranked_terms = [("term1", 3.5), ("term2", 7.1), ("term3", 5.2)]
sorted_terms = sorted(ranked_terms, key=lambda x: x[1], reverse=True)
print(sorted_terms)
```

Exercise4 – Query expansion metrics

We need to execute the previous exercise having two parameters: n & m .

- ➔ 'n' values in range [1, 5]
- ➔ 'm' values in range [3, 5]

Execution

These variables represent both the number of documents to return for the original query and the top terms that are used to expand the query.

For each pair (m , n) we expand the query and calculate the metrics of the expanded query as we did in exercise 2. Then, in each run the metrics are stored into a JSON file.

Note that, when invoking the retriever we passed 'n' as maximum results for the original query but then for the expanded one we pass 100. This is done to avoid query drifting when working with such small values when expanding the query.

When the program finishes its execution, the whole metrics are printed as a summary by the console, so the user can visualize them. They are also stored in JSON files in the directory "results-ex4". Also, note that we reuse a lot of code from previous exercises.

Conclusions

We are going to discuss how the values (n , m) have affected the performance of the system, considered that:

- 1) n is the number of documents to retrieve
- 2) m is the number of terms to expand the queries with

1. Effect of Parameter n (Number of Initial Documents Retrieved)

Parameter n controls the number of documents retrieved in the initial stage to identify relevant terms for query expansion. From the results:

a) **General Trend:** for a fixed value of m , increasing n initially improves the Macro F1 score, but beyond a certain point, the improvements decrease or even reverse slightly. This suggests that higher values of n retrieve more documents, including more relevant ones, but also increase the chances of incorporating noise (less relevant documents), which ultimately hurts accuracy.

2. Effect of Parameter m (Number of Terms Added to the Query)

The parameter m defines how many additional terms are added to the original query after analysing the initial documents.

a) General Trend: for any value of **n**, increasing **m** improves performance, as observed in the **Macro F1**, **Micro F1**, and **Macro Precision** metrics. Adding more relevant terms to the query helps refine the search and expand the coverage of the query, thereby enhancing overall performance.

Comparison table

Here, you can visualize the results for the different n, m values:

Configuration	Macro F1	Micro F1	Macro Precision	Macro Recall
n=1, m=3	0.1082	0.1035	0.0657	0.3090
n=1, m=4	0.1075	0.1026	0.0651	0.3054
n=1, m=5	0.1079	0.1022	0.0648	0.3018
n=2, m=3	0.1091	0.1042	0.0661	0.3064
n=2, m=4	0.1103	0.1049	0.0666	0.3057
n=2, m=5	0.1104	0.1047	0.0664	0.3044
n=3, m=3	0.1087	0.1047	0.0664	0.3069
n=3, m=4	0.1099	0.1052	0.0667	0.3075
n=3, m=5	0.1100	0.1050	0.0666	0.3055
n=4, m=3	0.1083	0.1045	0.0663	0.3088
n=4, m=4	0.1093	0.1054	0.0669	0.3071
n=4, m=5	0.1109	0.1062	0.0674	0.3095
n=5, m=3	0.1089	0.1053	0.0668	0.3103
n=5, m=4	0.1093	0.1056	0.0670	0.3083
n=5, m=5	0.1109	0.1060	0.0673	0.3068

As a conclusion, we have as best configuration the combinations (n=4, m=5), and (n=5, m=5) which offers the best overall performance, with the highest Macro F1 value (0.1109).

Also, increasing n initially improves performance by including more relevant documents, but very high levels introduce noise. On the other hand. increasing m (number of terms added) results in more focused queries and hence better performance.

The recommendation we can extract from this experiment is to use moderate values of n as much terms as we can to maximise precision and recall without degrading query focus.

Exercise5 – Comparison against Semantic Search

We are required to compare the best BM25 configuration plus query expansion with the results of applying a semantic retrieval system using a vector database and a pre-trained word embeddings model as we have seen in the laboratory notebooks.

Setup & Execution

For this program I have adapted the notebook we have seen in the laboratory session to our specific collection. Also, note that, for saving time, the chromadb collection is download from google drive and decompressed, but, if you want to create it, you can un-comment the function 'create_collection' call in the main method.

URL to the file:

https://drive.google.com/uc?id=1tme7R3L8QYt8kzIDwzIpayixbl0_mA-U

Comparative results

Configuración	Semantic search	Lucene with Query Expansion
Macro Precision	0.02	0.067
Macro Recall	0.0881	0.310
Macro F1	0.0511	0.111
Micro Precision	0.02	0.067
Micro Recall	0.0744	0.251
Micro F1	0.0315	0.106

Comparative Analysis

Precision (Macro and Micro)

Lucene outperforms semantic retrieval in terms of precision (macro and micro), by a significant margin. This suggests that Lucene is more effective in returning relevant documents relative to the total results retrieved.

Interpretation: the high precision of Lucene is explained by its reliance on exact lexical matches, which reduce noise in the results, especially in a technical domain.

Limitations of ChromaDB: semantic retrieval may be including documents that are conceptually similar, but not directly relevant, reducing precision. This could be an **out-of-domain problem**, where the semantic model is not adapted to the biomedical domain.

Recall (Macro and Micro)

Lucene also performs better in terms of the recall they produce.

Interpretation: Lucene can capture a larger number of relevant documents due to query expansion, while the semantic model may not be fully tuned to the biomedical domain, which limits its ability to retrieve relevant documents.

Limitation of Lucene: Despite being better than ChromaDB, Lucene's retrieval is still low, suggesting that both methods have difficulties in comprehensively covering the relevant documents in the corpus.

F1 Score (Macro and Micro)

The F1 score also favours Lucene, compared to semantic retrieval.

Interpretation: Lucene achieves a better balance between precision and recall, excelling especially in scenarios where lexical matches are critical.

Semantic retrieval, while offering some flexibility in general queries, does not achieve competitive performance on this dataset.

Reflection on Results

Lucene

Superior Technical Performance: In this specific domain, Lucene excels because most queries require exact matches or derivations of key terms.

Query Expansion: Query expansion improves retrieval by including related terms, which contributes to better balance in metrics.

Efficiency: It is more efficient in terms of execution time for large corpus, as it does not require computationally intensive operations such as embedding generation.

Limitations of Lucene depends on lexical matches and may fail when queries use synonyms or more abstract language. May be less adaptable for domains where language is ambiguous or less structured.

Semantic Retrieval

1) Inferior Performance in the NF-Corpus Domain: The paraphrase-multilingual-MiniLM-L12-v2 model is not specifically tuned for the biomedical domain, which negatively affects both precision and retrieval. Semantic retrieval models, like the paraphrase-multilingual-MiniLM-L12-v2, are not tailored to highly specialized domains and may suffer from reduced performance due to out-of-domain challenges.

Individual queries analysis

Semantic search will provide better results for those queries that:

- 1) Queries that benefit from semantic flexibility: embeddings models understand that terms do not have to be exact to be conceptually relevant. Synonyms and linguistic variations are not an issue, as the model can capture the semantic relationship between terms.
- 2) Queries that use terms or phrases related to broad concepts: the semantic model is effective in identifying related topics or concepts, even when the exact words do not match. This is especially useful in domains such as health, where the same topics may be treated with different terminology.
- 3) Queries that contain explicit context or include complex details: a semantic search system benefits from the context and overall understanding of the query, improving the ability to retrieve relevant documents that address complex or detailed topics.

Semantic search will provide worst results for those queries that:

- 1) Very general or ambiguous queries:
- 2) Queries with grammatical errors or vague formulation
- 3) Extremely specific queries: if a topic is very specialized the semantic search can fall in an out-of-domain issues
- 4) Queries with too many keywords or complexity: semantic models can benefit from clear and well-formulated queries, but when the query contains too many details or variants of terms, the model may lose focus or return too many scattered results.

Final Conclusion

For the NF-Corpus domain, Lucene with query expansion is clearly superior due to the technical nature of the corpus and queries. Semantic retrieval has potential in more eneral or multilingual domains, but in specific technical domains, Lucene remains more robust.

ANNEX

1) The following prompt was entered to ChatGPT to know how gdown could be used in Python to download and decompress a file stored in Google Drive:

Prompt: 'Hi ChatGPT, I need to download a .tar file from Google Drive using Python. I want to use gdown to do so, but I'm not sure how to use it. Once I download the file, how can I unzip or decompress it in Python? Can you explain to me the steps?'

ChatGPT answer:

ChatGPT's response:

Hi! Yes, you can use the `gdown` library to download files from Google Drive. Here's how you can do it step by step:

1. **Install gdown:** If you haven't installed `gdown` yet, you can install it using pip:

```
bash
pip install gdown
```

2. **Download the .tar file:** To download the file from Google Drive, you need the file ID. You can find this ID in the URL of the file. For example, in the link `https://drive.google.com/file/d/1GZ-EmkJKFm_bNUogg_8lbdqMuIjk1CDo/view?usp=sharing`, the file ID is `1GZ-EmkJKFm_bNUogg_8lbdqMuIjk1CDo`.

You can then use `gdown` in your Python script to download the file like this:

```
python
import gdown

url = 'https://drive.google.com/uc?id=1GZ-EmkJKFm_bNUogg_8lbdqMuIjk1CDo'
output = 'your_file.tar' # Choose your output file name
gdown.download(url, output, quiet=False)
```

3. **Decompress the .tar file:** Once the file is downloaded, you can use the `tarfile` module to extract its contents. Here's how you can decompress the `.tar` file:

```
python
import tarfile

# Open and extract the .tar file
with tarfile.open('your_file.tar', 'r') as tar:
    tar.extractall(path='path_to_extract_to') # Specify the directory to extract to
```

If you want to extract the file in the current directory, you can just call `tar.extractall()` without specifying a path.

4. **Complete example:** Here's the full Python code to download and extract a `.tar` file from Google Drive:

```
python
import gdown
import tarfile

# Step 1: Download the .tar file
url = 'https://drive.google.com/uc?id=1GZ-EmkJKFm_bNUogg_8lbdqMuIjk1CDo'
output = 'your_file.tar'
gdown.download(url, output, quiet=False)

# Step 2: Decompress the .tar file
with tarfile.open(output, 'r') as tar:
    tar.extractall(path='path_to_extract_to') # or just use tar.extractall() for the cu
```

Let me know if you need further assistance!

OptionalEx – Hybrid Search with Reranking

In this exercise, we modified the hybrid search notebook from the lab to include a reranking step using the 'rerankers 0.5.3' package. The goal was to apply the reranking technique to the combined results from both lexical and semantic search engines and evaluate the impact of reranking on search result quality.

Setup & Execution

This script integrates results from two different search approaches: a **lexical search** using BM25 and a **semantic search** using embeddings. The process involves:

1. Submitting queries to both the **lexical** and **semantic** search engines.
2. Merging the results using **Reciprocal Rank Fusion (RRF)**, which combines the ranks of both search engines.
3. Applying **reranking** to further optimize the results.
4. Calculating evaluation metrics (Precision, Recall, F1 score) for the reranked hybrid results.
5. Saving the results to a Json file.

Changes made to the notebook

Lexical Search Implementation (BM25):

- **Installation of Required Libraries:** The notebook installs bm25s for performing lexical search.
- **Obtention of queries & relevance judgments:** reusing previous functions.

Semantic Search Implementation:

- **Installation of Required Libraries:** The chromadb library is installed for semantic search, along with sentence-transformers to utilize pre-trained models for sentence embeddings.
- **Embedding the Corpus:** The embeddings for the corpus are precomputed and saved to a file. These embeddings are loaded and used in the semantic search engine.
- **ChromaDB Setup:** A persistent ChromaDB client is initialized to manage the semantic search. A collection is created using a multilingual model (paraphrase-multilingual-MiniLM-L12-v2), and the similarity metric is set to cosine distance.
- **Semantic Search:** Queries are passed to the ChromaDB collection, and the nearest semantic results are retrieved using pre-trained embeddings.

Query Submission Functions:

- **Lexical Search Queries:** The 'submit_queries_to_lexical_and_get_run' function is used to handle query submission to the lexical search engine. It performs query expansion for the best configuration of the exercise 2 and the best parameters of the exercise 4. Then, it returns the run collection.
- **Semantic Search Queries:** Similarly, 'submit_queries_to_semantic_and_get_run' is defined to handle submission of queries to the semantic search engine. Results are retrieved based on the precomputed embeddings.

Hybrid Search Approach:

- **Reciprocal Rank Fusion (RRF):** A custom function 'reciprocal_rank_fusion' is defined to combine the top results from both the lexical and semantic search engines. It merges the rankings using the Reciprocal Rank Fusion method, which assigns higher scores to documents ranked higher in either of the two rankings.
- **Combining Results:** The 'mix_lexical_semantic_runs' function is implemented to take the results from both the lexical and semantic searches, apply RRF, and produce a final mixed ranking. The results are stored with document IDs and their corresponding scores.

Main Execution Flow:

- **Files Loading:** the necessary file is loading reusing previous functions.
- **Search Execution:** Queries are submitted to both lexical and semantic search engines.
- **Result Mixing:** The results from both search engines are mixed using RRF, and the results are evaluated based on precision, recall, and F1-score metrics.
- **Metrics Calculation and Saving:** The evaluation metrics for the hybrid search are calculated and saved for further analysis.

Reranking Methods Tested

Original Search Results

We first obtained the results from both the lexical and semantic search engines independently. Then, the results were merged using Reciprocal Rank Fusion (RRF).

Reranking Step

After performing the RRF, we used the rerankers package to refine the results based on predefined ranking features. The reranker evaluates the relevance of documents in the merged list and reorders them for better quality.

Impact on the quality of the result

Table of values:

Configuración	Hybrid Search
Macro Precision	0.0604
Macro Recall	0.2818
Macro F1	0.1011
Micro Precision	0.0604
Micro Recall	0.2249
Micro F1	0.0953

Summary and Conclusion

Hybrid Search has an overall lower performance compared to Lucene with Query Expansion, but it improves semantic search in all key metrics (precision, recall, F1).

Precision and Recall are lower in the hybrid approach, resulting in a lower F1 Score as well. The Hybrid Search approach appears to be extracting several irrelevant documents, which decreases both precision and the ability to recall relevant documents. Neither Semantic search nor Hybrid search can obtain better results than the ones obtained for Lucene with stop words removal and stemming application with query expansion.

This is, probably, due to the great technical aspect of our collection, where we have very specific and technical documents (and queries). In this context, Semantic search does not shine as much as it would to in more general context (out-of-domain issue).

Set of screenshots of the program

```
if __name__ == "__main__":
    try:
        queries = load_queries()
        relevance_judgements = load_relevance_judgements()
        stemmer = Stemmer.Stemmer("english")

        collection = connectToDB("TREC-COVID_collection", "./chromadb-storage")

        ...

        We submit the queries to the lexical and semantic search engines.
        ...

        original_lexical_run = submit_queries_to_lexical_and_get_run(
            queries, stemmer, max_results=100)
        original_semantic_run = submit_queries_to_semantic_and_get_run(
            queries, collection, max_results=100)

        ...

        We mix the lexical and semantic runs.
        ...

        mixed_run = mix_lexical_semantic_runs(original_lexical_run, original_semantic_run)
        metrics = calculate_metrics(mixed_run, relevance_judgements)

        # Save the results to a JSON file in the specified directory
        save_results(mixed_run, relevance_judgements, "results-optional-ex", "hybrid_run_results", metrics)

    except Exception as e:
        print("An unexpected error occurred.")
```

Conclusions of the project

During this project we have reviewed both the theory concepts explained in class and the practical application of those concepts that we saw in the laboratory sessions.

We have tried different approaches to maximize the results of searching with queries inside a collection, exploring not only normal lexical search, but also query expansion (learning how to use the Signed Root LLR algorithm), with Semantic Search and a vector database and, finally, with a Hybrid approach between Lexical & Semantic search

This project has helped to understand how the retrieval of information is performed and how, depending on the technique used to retrieve information, you can obtain more efficient results, which, in the end, results into a better system for the user.

Adrián Martínez Fuentes, UO295454