README

Assignment 1 (CSI-4107)

Members of Group 25

Member	Student Number
Nalan Kurnaz	300245521
Alona Petrova	300074852
Kevin Luong	300232125

Contributions

Member	Contributions
Nalan Kurnaz	- Worked on implementing Inverted Index (indexing.py) - Report writing
Alona Petrova	- Implementation of retrieve_and_rank and the first version of main.py - Implementation of naive ranking based on the tf-df cosine similarity - Report writing
Kevin Luong	- Worked on the pre-processing - Implemented BM25+ - Investigate synonym replacement and pseudo-relevance feedback feasibility - Work on report

Functionality

Our Information Retrieval (IR) system is designed to retrieve and rank documents based on a given query. The system follows these key steps:

Preprocessing

Documents undergo tokenization, stopword removal, and stemming/lemmatization to standardize terms.

Indexing Phase

An inverted index is built, mapping terms to document IDs along with their term frequencies.

Ranking and Retrieval

The input query is preprocessed similarly to the documents (tokenization, stopword removal, stemming).
The system calculates similarity scores between the query and each document using BM25+ scoring. It then sorts the documents based on their scores and returns the top 100 most relevant documents.

Get Started/Instructions on how to run the system

1. Install Python: Ensure you have Python installed on your local machine. You can download it from Python's official website.
2. Download trec_eval: The trec_eval tool can be downloaded from TREC's official website.
3. Extract the trec_eval package: The downloaded file is a .tar archive. Use an extraction tool such as tar (Linux/macOS) or 7-Zip (Windows) to extract it.
4. Ensure both trec_eval and your code are in the same directory for easier execution.
5. Compile trec_eval
    - On POSIX systems:
      `cd trec_eval-9.0.7 make`
    - On MinGW/GCC:
      `gcc -o trec_eval trec_eval.c`
6. Install all the necessary dependencies in the `requirements.txt` file using `pip install`.
7. Run the python script `python main.py`
8. Evaluate results by using trec_eval (you must copy over the scifact/qrels/test.txt and <bm25_result_file> into the same directory as trec_eval)
   `./trec_eval test.txt <bm25_result_file>`
   Replace <bm25_result_file> with the name of your BM25 result file (e.g., bm25_result_for_titles.txt).

Analysis of Algorithms, Data Structures, and Optimizations

In this section, we provide information on the algorithms and data structures used. Additionally, we will discuss the optimization steps taken to improve out system.

Algorithms

Pre-processing

To extract index terms and count term frequencies, the algorithm follows:

- Lower and strip the original text.
- To get rid of any unicode characters, we need to first encode and decode the characters in order to properly remove them. Otherwise, they are stored in the string as actual unicodes such as "\u2013".
- Split the text using a regex word splitter that splits on spaces and punctuation, but maintains hyphens. The punctuation will be removed later.
- Remove all non-letter characters (except for the hyphen).
- For terms that have a hyphen, determine if it is a compound word connected by a hyphen or a word composed of an affix and a root word. If it is the former, split the word into 2 terms. If not, remove the hyphen and keep the word as 1 term.
- Count the term frequencies of each term and return them as a dictionary with the term as key and the term frequency as the value.
- Remove all stopwords from index terms by using a set difference.
- Lemmatize each term. If, after lemmatization, the term happens to match an existing term (e.g. "flies" lemmatizes to "fly"), then combine their term frequencies.
- Return the dictionary of index terms and term frequencies.

Indexing Phase (Building the Inverted Index)

The indexing phase processes documents and builds an inverted index, which maps terms to the documents they appear in. The steps include:

- Extracting index terms using tokenization, stopword removal, and lemmatization.
- For each document, extract terms and update their frequencies in the inverted index.
- Storing term frequencies for later ranking adjustments.
- Enabling quick lookup of document postings for each query term.

Retrieve and Ranking

- The algorithm requires the following parameters:
  - query (str): The query for which the documents are being ranked.
  - inverted_index: The index of words and the documents that contain them.
  - documents: The set of documents to be ranked.
  - avg_doc_length: The average document length used in BM25 calculation.
  - k1, b, delta: BM25 hyperparameters that adjust the score calculations.
  - top_n: The number of top documents to return.
- Initialize the similarities dictionary: This will store the similarity scores for each document

- For each document, the code checks whether it contains at least one word from the query. It loops over the query terms and checks if they are present in the inverted index. If the document contains any query word, it proceeds to calculate the BM25 score.mportant for ranking documents based on query relevance.
- If a document contains at least one query term: calculate the BM25 score for this document. Retrieve the BM25 vector for the query and the document using the function get_bm25_query_vector (this computes the BM25 score for each term in the query, considering the document's length, the average document length, and the BM25 hyperparameters).
- Once the BM25 query vector is obtained, the code computes the cosine similarity between the query vector and the precomputed document vector. If the similarity score is greater than zero, it is added to the similarities dictionary with the doc_id as the key.
- Sort the similarities dictionary by the similarity score in descending order to rank the documents.
  If no document has a similarity score greater than zero, print a message indicating that no documents were returned for the query.
- After sorting, retrieve the top n documents based on the highest similarity scores and return them as a list of tuples containing the doc_id and its corresponding similarity score.

To calculate the BM25+ weighting of each term, the BM25 formula by Robertson and Sparck-Jones (1976) was modified by adding a new parameter, delta:

$$\text{weight} = \frac{(\text{term\_freq} + \delta) \times \log\left(\frac{\text{total\_documents} - \text{doc\_freq} + 0.5}{\text{doc\_freq} + 0.5}\right)}{(k_1 \times ((1 - b) + (b \times \text{doc\_length}/\text{avg\_doc\_length})) + \text{term\_freq}}$$

Data Structures

Pre-processing

For the documents and queries, we created a parent class called RetrievalItem and had a class Document and Query inherit that class. By doing this, we only needed to define variables like IDs and text/queries once. At initialization of a new document of query, all of the information pertaining to its ID, text content, and index terms with its term frequency are all stored in the object, so we do not need to compute the index terms and term frequency each time. Furthermore, this will facilitate creating and indexing an inverted index because the terms and term frequencies will always be stored within the object.

To store the index terms within each Document and Query object, the index terms are stored as a dictionary in which the key is the index term and the value is the term frequency for the document/query.

In addition, we have used the following data structures to improve perofrmance of the system:

- `set`: set is used for the stop_words to facilitate fast lookups. It allows for constant time complexity (O(1)) when checking if a word is a stop word, which is crucial for efficiently filtering out common words
- `RegexpTokenizer` (from nltk.tokenize): This tokenizer is used to split the text into words based on a regular expression. It handles punctuation and hyphenated words, ensuring that terms like "pre-diabetes" are split properly into meaningful components.
- `Counter` (from collections): The Counter is used to count the frequency of each term in the text. This is an efficient way to track the number of occurrences of each unique word in the document.
- `dict`: A dictionary is used to store the index terms (after processing) along with their term frequencies. This data structure allows easy access and modification of the term frequencies.
- `list`: After lemmatization, the synonyms or terms are stored in a list. This allows easy iteration over each word for further processing, such as removing non-alphabetic characters or filtering out empty strings.

Indexing Phase (Building the Inverted Index)

- `Nested dictionaries` for Inverted index: `{term → {doc_id → frequency}}` (nested dictionaries). Maps each term to a dictionary of document IDs and their corresponding term frequency, enabling efficient term look-up across documents.
- `Defaultdict` (defaultdict(lambda: defaultdict(int))) avoids handling missing keys manually.
- `List of tuples` (each containing `doc_id` and `frequency`): Each term has a posting `list`, containing `doc_id` and frequency, enabling fast retrieval.

Retrieve and Rank

- `Query` object: Represents the search query and provides methods to access the terms included in the query.
- `Inverted Index`: Maps terms (words) to the set of documents that contain those terms.The query terms are checked against the inverted index to determine which documents contain the query terms. This helps in filtering out documents that don’t match the query.
- `Document`: Stores the documents to be ranked. Each document is identified by a unique doc_id.
- `dictionary`: The dictionary stores computed similarity scores between the query and each document as well as the precomputed vector representation of each document. It allows fast lookup by using doc_id as the key, ensuring efficient retrieval and ranking of relevant documents.

Optimizations

Pre-processing

- When splitting words, we used the NLTK Regex Word Splitter instead of opting simply for Python's string split function. This is to avoid cases where terms may be stuck together because of a missing space after an ending punctuation mark (e.g. "class.The" should be 2 terms). Therefore, we decided to use regex to split based on punctuation marks and spaces, but maintain hyphens because of compound words.
- While exploring the documents, we found that certain terms that were hyphenated may actually be a compound word whereas some were simply words with prefixes. In the case of terms like "body-mass" which is a compound word, we found that it would sometimes appear as "body mass" as well. To ensure an accurate term frequency and document frequency, we created a check that would split hyphenated words into parts and then check for each part to see if it was an actual word on its own using the Spellchecker library. If each part is a word, we would split them into their own words. Otherwise, we kept them as a single term and removed the hyphen.
- For word stemming, we opted for lemmatization because it left room for future integration with synonym replacement. Since stemming would simply cut the end off of a word, the result could be a non-word (e.g. "flies" becomes "flie" which is not a word). By using lemmatization, we were able to output a word that could be fed into a domain-specific thesaurus like MeSH. Unfortunately, we could not gain access to a dictionary API without guaranteeing that 1) the API would always work and 2) that the API key would still be valid. Therefore, we left room for, if that were to happen, we could still use a thesaurus for synonym replacement. On the other hand, using WordNet, though local, does not contain enough domain-specific synonyms to best match a word in the inverted index and the word in the query.

Ranking

- When calculating the weight for each term, we opted for BM25, specifically the BM25+ variant, instead of TF-IDF. This is because BM25 considers factors like document length and can be fine-tuned using its hyperparameters k1 and b. We found that document length may vary which, to improve performance, needs to be considered in our ranking algorithm. We then chose the BM25+ variant that address the issue with BM25 where long documents that match a query term may be ranked the same as shorter documents that do not contain a query term at all. Therefore, we introduced the delta hyperparameter to control this behaviour. As a result of our testing, we have found that it has improved our MAP.
- We explored the possibility of using synonym replacement using WordNet, but found that MAP was decreasing. We found that certain queries did not match with the correct

document because the query would contain terms that did not exactly match any term in the document. To circumvent this issue, we wanted to see if we could try substituting query terms that had 0 term frequency within the document with a synonym that appeared at least once in the document. The issue here is that 1) a lot of the query terms are domain-specific, so WordNet is unable to find the exact synonym and 2) oftentimes, generic terms were overfit to the document and therefore artificially inflated its similarity score. Therefore, this had caused a decrease in our MAP.

- We also explored the possibility of using a pseudo-relevance feedback loop. However, we found that our MAP was also decreasing. This follows the principle of "trash-in-trash-out" because if our top results were not relevant enough, then performing a feedback loop of the top n documents would simply fit our query to return more irrelevant documents. This resulted in a drop in our MAP.

Results

In this section, we present the results obtained after running our system.

RUN 1: Title and text

In this experiment, we constructed an inverted index using both the title and text of documents and retrieved rankings for all test queries. The top 10 retrieved documents, along with their similarity scores, are provided for queries 1 and 3. The complete ranking data is available in `bm25_result_for_titles_and_text.txt`.

Query 1: "0-dimensional biomaterials show inductive properties."

Query 3: "1,000 genomes project enables mapping of genetic sequence variation consisting of rare variants with larger penetrance effects than common variants."

Results for Query 1: top 10 docs

Query ID	Q0	Document ID	Rank	Similarity Score	Run
1	Q0	24998637	1	0.193820	run1
1	Q0	13231899	2	0.191900	run1
1	Q0	10931595	3	0.168496	run1
1	Q0	9580772	4	0.160966	run1
1	Q0	16939583	5	0.160644	run1
1	Q0	803312	6	0.160289	run1
1	Q0	31543713	7	0.159871	run1

Query ID	Q0	Document ID	Rank	Similarity Score	Run
1	Q0	17123657	8	0.148462	run1
1	Q0	10607877	9	0.148276	run1
1	Q0	20758340	10	0.146422	run1

Result for Query 3: top 10 documents

Query ID	Q0	Document ID	Rank	Similarity Score	Run
3	Q0	2739854	1	0.355093	run1
3	Q0	39661951	2	0.323758	run1
3	Q0	19058822	3	0.317263	run1
3	Q0	4632921	4	0.288806	run1
3	Q0	23389795	5	0.276622	run1
3	Q0	4378885	6	0.273045	run1
3	Q0	1544804	7	0.267283	run1
3	Q0	17702490	8	0.261034	run1
3	Q0	4414547	9	0.259740	run1
3	Q0	1067605	10	0.254098	run1

For Query 1, the highest-ranked document (ID: 24998637) has a similarity score of 0.193820, indicating it is the most relevant match.
However, the actual document the query originates from (ID: 31715818) was not found in the top 10 results. Further investigation showed that it was not included in the top 100 retrieved documents for Query 1.

For Query 3, the top-ranked document (ID: 2739854) has a similarity score of 0.355093, making it the most relevant result for this query.
However, the actual matching document according to test.tsv (ID: 14717500) was ranked 13th by our system

RUN 2: Only title

In this run, we used only title for inverted index and retrieved the ranking for all the test queries. The top 10 documents with similarity scores are provided for queries 1 and 3. The rest of the ranking data can be found in `bm25_result_for_titles.txt`

Results for Query 1: Top 10 Documents

Query ID	Q0	Document ID	Rank	Similarity Score	Run
1	Q0	4459491	1	0.511560	run2
1	Q0	25950264	2	0.473070	run2
1	Q0	10029970	3	0.451824	run2
1	Q0	18953920	4	0.436798	run2
1	Q0	9580772	5	0.426131	run2
1	Q0	25404036	6	0.393896	run2
1	Q0	11674288	7	0.382282	run2
1	Q0	31543713	8	0.359190	run2
1	Q0	38037690	9	0.345779	run2
1	Q0	15593561	10	0.344601	run2

Results for Query 3: Top 10 Documents

Query ID	Q0	Document ID	Rank	Similarity Score	Run
3	Q0	2739854	1	0.792977	run2
3	Q0	23389795	2	0.710271	run2
3	Q0	14717500	3	0.672403	run2
3	Q0	4421746	4	0.569425	run2
3	Q0	52850476	5	0.538769	run2
3	Q0	6077214	6	0.533804	run2
3	Q0	13519661	7	0.522262	run2
3	Q0	10279084	8	0.521175	run2
3	Q0	17088791	9	0.518665	run2
3	Q0	4378885	10	0.515995	run2

For Query 1, the highest-ranked document (ID: 4459491) has a similarity score of 0.511560, but the actual document (ID: 31715818) was not found in the top 10 or even the top 100 retrieved documents.

For Query 3, the top-ranked document (ID: 2739854) has a similarity score of 0.792977, while the actual matching document (ID: 14717500) was ranked 3rd by our system.

For Query 1, neither approach retrieved the actual document in the top 10, indicating that title and text indexing may not capture its relevance effectively.
For Query 1, while some documents appear in both lists (e.g., IDs 31543713, 9580772), others are unique to each method. This suggests that using only titles retrieved different documents that were not picked up when using both title and text.

It important to note, that the absence of a match for Query 1 may be due to the short query length and lack of exact term overlap with the document, as searching word by word did not yield matches. The document may also use different terminology or phrasing, making it harder for the retrieval system to establish relevance.

For Query 3, although several documents overlap (e.g., IDs 2739854, 23389795, 4378885), the ranking order differs, and some documents retrieved using only titles were not retrieved using title + text. Notably, the actual relevant document (ID: 14717500) ranked higher in the title-only approach (3rd place) than in the title + text approach (13th place), suggesting that relying on titles may improve precision in some cases.

Vocabulary

The vocabulary of dataset contains 31605 unique tokens extracted from the indexed documents. The vocabulary was generated after preprocessing steps such as tokenization, stopword removal and case normalization.

The sample of the top 100 most frequent tokens from our vocabulary is shown below:
'cell', 'result', 'study', 'increase', 'protein', 'suggest', 'factor', 'associate', 'gene', 'role', 'expression', 'human', 'control', 'disease', 'effect', 'function', 'patient', 'data', 'level', 'identify', 'mechanism', 'conclusion', 'induce', 'method', 'analysis', 'model', 'response', 'specific', 'demonstrate', 'type', 'activity', 'treatment', 'compare', 'target', 'development', 'signal', 'cancer', 'reduce', 'require', 'regulate', 'report', 'process', 'base', 'pathway', 'receptor', 'significantly', 'change', 'finding', 'indicate', 'involve', 'present', 'risk', 'cause', 'potential', 'remain', 'age', 'clinical', 'mice', 'activation', 'determine', 'reveal', 'system', 'complex', 'relate', 'group', 'evidence', 'population', 'develop', 'tissue', 'bind', 'measure', 'express', 'number', 'mediate', 'observe', 'year', 'form', 'dna', 'decrease', 'know', 'growth', 'vivo', 'dependent', 'background', 'review', 'outcome', 'tumor', 'objective', 'early', 'lead', 'regulation', 'investigate', 'test', 'activate', 'interaction', 'occur', 'cellular', 'molecular', 'design', 'major'

In this part, we have extracted the 100 most frequent tokens from vocabulary. These tokens seem related to medical and scientific fields especially focusing on brain development, diffusion MRI scans and neuroscience-related research. For instance, we can see words like ' cell', 'protein', gene’, and ‘disease’ are commonly found in medical literature. On the other hand, words like 'analysis' or 'model' would refer to statistical evaluations. One of the observations from the extracted vocabulary is that stop words (e.g.'the', 'is' or 'and') don't appear in the list, meaning that proper preprocessing techniques were implemented. Also, all words appear to be in lowercase, indicating that text normalization was performed to make sure tokenization is consistent.

Output & Evaluation: Mean Average Precision (MAP)

Document Content	MAP (BM25+)
Titles	0.2958
Titles + Text	0.5485

The effect of evaluating more queries are observed in the BM25+ MAP (Mean Average Precision) scores which are calculated for both titles only queries, and titles and text queries. When evaluating only titles, the MAP score was 0.2938, whereas assessing with both titles and full text increased the MAP score to 0.5485. This means that full-text indexing enhances document retrieval performance compared to queries with only titles. One of the reasons for this improvement is that titles are short and concise, and they don't provide much context for BM25+ to find relevant matches. However, full documents have more vocabulary and detailed description, leading BM25+ to have more accurate result in term frequency and inverse document frequency in the ranking of documents. Also, full-text indexing increases the probability of matching queries with relevant terms since it has more opportunity to have term overlap and semantic variations. Therefore, having more contextual information improves retrieval performance.

Conclusion

In summary, the extracted vocabulary we obtained has more context in medical and scientific terms. Observations from the vocabulary confirm that preprocessing techniques such as stop word removal and lowercase normalization were successfully applied. The BM25+ MAP score difference refers that the full-text indexing leads to better document ranking and retrieval performance as it provides more contextual information.