Assignment #1

Information Retrieval System

CSI 4107

Group 9 members:

Ousmane Barry (300221573)

Quoc Dat Phung (300164087)

Rami Slimane-Kadi (300237431)

Date : 2025-02-09

Introduction

This is a report for the information retrieval course’s assignment 1. In this assignment we were tasked with creating a basic IR system (information retrieval system) which takes a set of queries located in a json file and a corpus of documents also located in a json file. It then loops through each query, performs some basic IR tasks/calculations and returns a list of the top 100 documents (in terms of similarity) associated with the respective query. The top 100 documents are returned in ascending order. For instance, for query1, its top document will have the highest cosine similarity score and its 100th document will have the lowest. In this report we will discuss the contributions of each team member, the general requirements needed to run our main.py file, the details of each step and function required to complete this assignment, and examples of the results we had (results in results.txt and MAP scores).

Group Member contributions

Name	Task
Quoc Dat Phung	- Code for Step 1 - Code for Step 2 (up to the building inverted index part) - Report
Ousmane Barry	- Write function to run all queries - Write function to calculate map value - Produce all 4 results files - Report
Rami Slimane-Kadi	- Built max_frequency dictionary and converted inverted ind. to weighted inverted index (Step 2) - Worked on query_vector_maker_and_retrieval and ranking functions (Step 3) - Report

How to run

- pip install -r requirements.txt
- py main.py
- py calculate_map.py

Can take about 30 seconds

Explanation of the Algorithms

There are three parts to building the information retrieval system in this assignment: preprocessing, indexing, and retrieval/ranking.

Preprocessing

For the preprocessing function, the algorithm performs six steps. The first step is tokenization, which splits the text into words. For example, if the input string is “I love reading books in 2025!”, tokenization results in the list ["I", "love", "reading", "books", "in", "2025", "!"]. The second and third steps convert all strings in the list to lowercase and remove any punctuation. At this point, the list becomes ["i", "love", "reading", "books", "in", "2025"]. The fourth step removes any stop words, such as “the”, “in”, “on”, etc. The professor provided a list of stopwords in the file stopwords.txt. The fifth step removes any numbers, such as “2025”, and the final step performs stemming, which converts words to their root forms. For example, the word “reading” becomes “read”. After all six steps, the input becomes ["love", "read", "book"]. Finally, we loop through each row in the file corpus.jsonl, combine the title and the text of the document into one string, and apply the preprocessing function on that string. The reason why the title is included in the string is because it provides important context about the document's content.

Indexing

The second part of the algorithm is about indexing. This is performed by the function build_inverted_index_from_corpus(). The goal is to create an inverted index which is a data structure that maps each unique word (or term) to the list of documents where that word appears. To do this, the function first creates an empty dictionary called inverted_index. The function iterates through each word in the processed text. If the word is not in the inverted index, it adds the word with the document ID and sets the count to 1. If the word exists but the document ID is new, it adds the document ID with a count of 1. If both the word and document ID are present, it increments the count for that document. This builds an inverted index that tracks word frequencies per document. There are a total of 18820 words in the vocabulary after the inverted index is created. We then realized that the inverted index holding raw counts had to be converted into the inverted index with token weights ($w_i = tf_i * idf_i$), we called this the weighted inverted index (WII). We created an exact copy of the inverted index using deepcopy() function in python (weighted_inverted_index= copy.deepcopy(inverted_index)). From there, we looped through the WII and computed both the tf and idf to find the weight for the specific token in a document. We then replaced the raw count of that token/term for that specific document with its weight.

Retrieval and Ranking

The retrieval and ranking were separated into three different functions. We had query_vector_maker_and_retrieval() and ranking() which took care of most the of system's logic/calculations, and the run_queries_and_write_to_file() function which ran all the previous functions, sorted the retrieved documents for the respective query, and wrote all the results to the file. The query_vector_maker_and_retrieval() function is responsible for creating a query vector by first calling the preprocessing function to generate the list of key words/tokens for the query. It then creates a dictionary/vector containing the tokens and their weights (which we calculate using TF-IDF weighting). The next task the function is responsible for is the creation of the retrieved_docs dictionary. This is the dictionary containing ALL documents that contains at least ONE of the query's terms as keys. Those documents (dictionary keys) point to another dictionary containing the intersection between the query terms and the doc terms (so all shared terms with weights). For instance, query1 has terms {A,B,C,D} so the retrieved docs dictionary will look like this: { doc1:{A:1.1, B:2.23}, doc2:{B:2.34, C:3.02}, doc3:{A:3.53, B:1.134, C:2.243} }. The function then returns a tuple containing both the query vector and retrieved docs.

Moving on to the ranking() function. This function takes the retrieved_docs, query_vector, and document_vectors (this is another dictionary created in step 2 containing ALL documents in the corpus, as keys, pointing to another dictionary containing ALL words found in the specific document/key and their associated weights) as parameters. This function will first calculate the euclidean norm of the query vector to find its length and hold it in a variable (this doesn't change since we're comparing one query with multiple documents). It will then loop through each document in the retrieved_docs dictionary and from there it will compute the dot product between the query vector and the current document's vector in the retieved_docs dictionary. The last step before computing the cosine similarity is the length of the current document's vector (which we calculate using the document_vectors dictionary). Once that is computer, we store the result in the results_dict dictionary containing the retrieved document and its cosine similarity score (between it and the query). We then return the result_dict.

The last function called run_queries_and_write_to_file() (technically it could be merged with the ranking() function but in order to have less confusing/bloated code, we decided to separate them) takes the result_dict dictionary and sorts the results in ascending order (highest score to lowest). It then writes the top 100 results/scores to a results.txt file in a specific format required by the prof.

Results Discussion

MAP Calculation Explained

The script first loads the qrels file to build a dictionary that maps each query ID to its corresponding document relevance scores. It then processes the results file in a similar manner, constructing another dictionary that maps each query ID to the document scores returned by the system. Using these two dictionaries, the script employs pytrec_eval to evaluate the retrieval performance by computing the Mean Average Precision (MAP) for each query based on a comparison between the system's scores and the ground-truth relevance judgments. Finally, it aggregates the individual MAP scores by averaging them across all queries and prints out the final overall MAP value.

results_top_100_title_text.txt :	0.4279
results_top_100_title.txt :	0.3340
results_top_doc_title_text.txt :	0.3009
results_top_doc_title.txt :	0.2569

Title only run

Uses only document titles for indexing. Limited vocabulary leads to lower recall (misses terms in document bodies).

Text and title run

Combines titles and full text. Higher recall and precision due to richer vocabulary from document bodies.

Hundred tokens from vocabulary

```
['cho', 'nonsteroid', 'dafachron', 'erg', 'cardinali', 'doi', 'scheme', 'theoret', 'across', 'pyy', 'cleans', 'multicopi', 'slam', 'unaid', 'hypochlorit', 'continu', 'hyaluronidas', 'wortmannin', 'rune', 'deterr', 'enlighten', 'psca', 'dxdx', 'hdi', 'hydra', 'kupffer', 'rhc', 'illeg', 'plagu', 'moreov', 'hormon', 'tmz', 'engrail', 'nonenzymat', 'vs', 'albeit', 'nimesulid', 'lineal', 'outgrowth', 'joint', 'babi', 'sickl', 'boophilu', 'dark', 'borrelia', 'flavopiridol', 'unfett', 'shi', 'finasterid', 'cystogenesi', 'overcomplet', 'echocardiographi', 'one', 'expos', 'indol', 'mutas', 'imv', 'pmhc', 'murdock', 'sidak', 'hub', 'symptom', 'complianc', 'transgener', 'dyadic', 'polyubiquityl', 'worthwhil', 'sibm', 'turbidimetri', 'gyras', 'ident', 'expend', 'olmst', 'legionella', 'felv', 'meningoenceph', 'pulmonologist', 'bartonella', 'bn', 'lpa', 'refold', 'hutzel', 'prune', 'uv', 'sh', 'pyoderma', 'neuroepithelium', 'needl', 'american', 'erythropoiet', 'pten', 'unidimension', 'flow', 'physiolog', 'nexu', 'filopodi', 'autocrin', 'vasorelax', 'tyramin', 'levofolin']
```

The vocabulary is 18820 words

First 10 answers of first 2 queries

```
query_id Q0 doc_id rank score tag
0 Q0 26731863 0 0.11689 tag_0_26731863
0 Q0 13231899 1 0.09986 tag_0_13231899
0 Q0 10906636 2 0.08762 tag_0_10906636
0 Q0 26071782 3 0.08458 tag_0_26071782
0 Q0 42421723 4 0.08156 tag_0_42421723
0 Q0 994800 5 0.08153 tag_0_994800
0 Q0 35008773 6 0.07582 tag_0_35008773
0 Q0 12156187 7 0.06786 tag_0_12156187
0 Q0 21439640 8 0.06745 tag_0_21439640
0 Q0 42731834 9 0.06477 tag_0_42731834
```

```
2 Q0 17333231 0 0.38617 tag_2_17333231
2 Q0 42240424 1 0.35234 tag_2_42240424
2 Q0 13734012 2 0.25886 tag_2_13734012
2 Q0 695938 3 0.17721 tag_2_695938
2 Q0 18617259 4 0.17532 tag_2_18617259
2 Q0 1292369 5 0.15525 tag_2_1292369
2 Q0 11880289 6 0.14595 tag_2_11880289
2 Q0 103007 7 0.13037 tag_2_103007
2 Q0 18340282 8 0.12978 tag_2_18340282
2 Q0 16398049 9 0.09288 tag_2_16398049
```

Conclusion

In summary, the implemented information retrieval system successfully integrates key IR components preprocessing, indexing, and retrieval/ranking to process queries and rank documents based on cosine similarity. The thorough preprocessing steps, including tokenization, stop word removal, and stemming, combined with a well constructed weighted inverted index, enabled the system to effectively capture document relevance. The experimental results, particularly the MAP scores, demonstrate that incorporating both titles and full-text data significantly enhances retrieval performance compared to using titles alone. This assignment not only met its objectives but also provided valuable insights into the practical challenges and benefits of various IR techniques.