Information Retrieval System

Team #4

CSI4107
Diana Inkpen

Assignment #1
Feb 9th, 2025

Tengyang Deng, 300156567
Michael O'Sullivan, 300228801
Yale Li, 200069299

Introduction

This report describes how to design and implement an Information Retrieval (IR) system based on the vector space model for a collection of documents. This report includes a description of the functionality, how to execute the program, included technical details, and finally an evaluation of its performance.

Work distribution

Members	Tasks
Tengyang Deng	Retrieval and Ranking, Query testing, Report work
Michael O'Sullivan	Preprocessing, Index Building, Report Work
Yale Li	Title Filtering + Index building, Report Work

Functionality Explanation

The program is designed in the Python language. Implement an indexing scheme based on the vector space model.

IR system class

The whole IR system is built in a class object and implemented in the IRsystem.py file. The class contains the functions class initialization, preprocessing a query, building an inverted index, and retrieving relevant documents / ranking.

Class Attributes

The IR systems class contains following attributes:

- stop_words (set): Set of stopwords for text cleaning during tokenization
  - Built from stopwords.txt file (from assignment) if the file exists.
- index (dict): Inverted index mapping words to document IDs
  - Format: [Word <string>, Document IDs <set()>]
- docs (dict): Stores cleaned document texts
  - Format: [DocumentID <string>, Cleaned Text <String>]
- vectorizer (TfidfVectorizer): TF-IDF vectorizer instance

The initialization of the class will read stopwords list in dict, if the file could not load, it will return a message that no stopwords are used. Then, we initialize the inverted index in the form of defaultdict set, which does not need to check the keys and simplifies the code. This class is created and its functionality is used in main.py during execution.

Preprocess query function

This function cleans and tokenized text by:

- Removing HTML tags
- Removing punctuation and numbers
- Lowercasing
- Removing stopwords

It reads the raw text input as an argument and finally returns a list of cleaned tokens

Build inverted index function

This function builds an inverted index by mapping each unique word across all the documents to a set of documents that contain it, and stores document texts for retrieval. It loads the corpus_file (str, path to file), as an argument. But it does not return anything since it only modifies the index and documents inside the IRsystem class object. For the execution, it goes through the corpus file, preprocesses each document (clean the text and title for tokens), and builds the inverted index. After the index is built, a message with the number of cleaned documents will be presented.

To fulfill the requirement that the index contains title only, we add a flag in the argument for the user to choose.

Retrieval and ranking function

This function Searches for relevant documents using TF-IDF and cosine similarity. It read query_text (str): Raw user query and top_k (int, default 100): Number of top-ranked results as arguments. Then it will return a list of ranked pairs as output. This function preprocesses the raw query, then finds and retrieves top_k relevant docs from cleaned documents using the inverted index and stores them in a dict object (return an empty list if no relevant docs are found). After retrieving the documents, it applies TF-IDF vectorization on those documents and query, then computes the cosine similarity for each. In this case, both TF-IDF and cosine similarity are calculated by using sklearn library (simple and easy to use). Finally, it ranks the documents by similarity in descending order and returns the top_k most similar documents to the query as {Document ID, Similarity Score}.

Save Results function

This function searches queries (only odd-numbered) and saves results in required format.

Args:
  query_file (str): JSON file containing test queries.
  output_file (str): Output file to save results.
  relevance_file (str): Path to the TREC relevance judgments file.
  top_k (int): Max results per query (set to 100 as default).

It first allows the user to enter the tag as an identifier, then, loop the query files in odd order and perform retrieval ranking for each of them. After that, it ranks the top 100 documents in required format for each query. At last, all the ranked list will be written into the output file (e.g. Results.txt)

Display sample function

To match the requirement, this function will displays:

1. Number of Vocabulary.
1. First 100 tokens from the vocabulary.
2. First 10 ranked results for the first 2 queries.

Args:
  query_file (str): JSON file containing test queries.
  top_k (int): Number of top-ranked results to display per query.

Basically perform the above functions to get results.

Execution Instruction

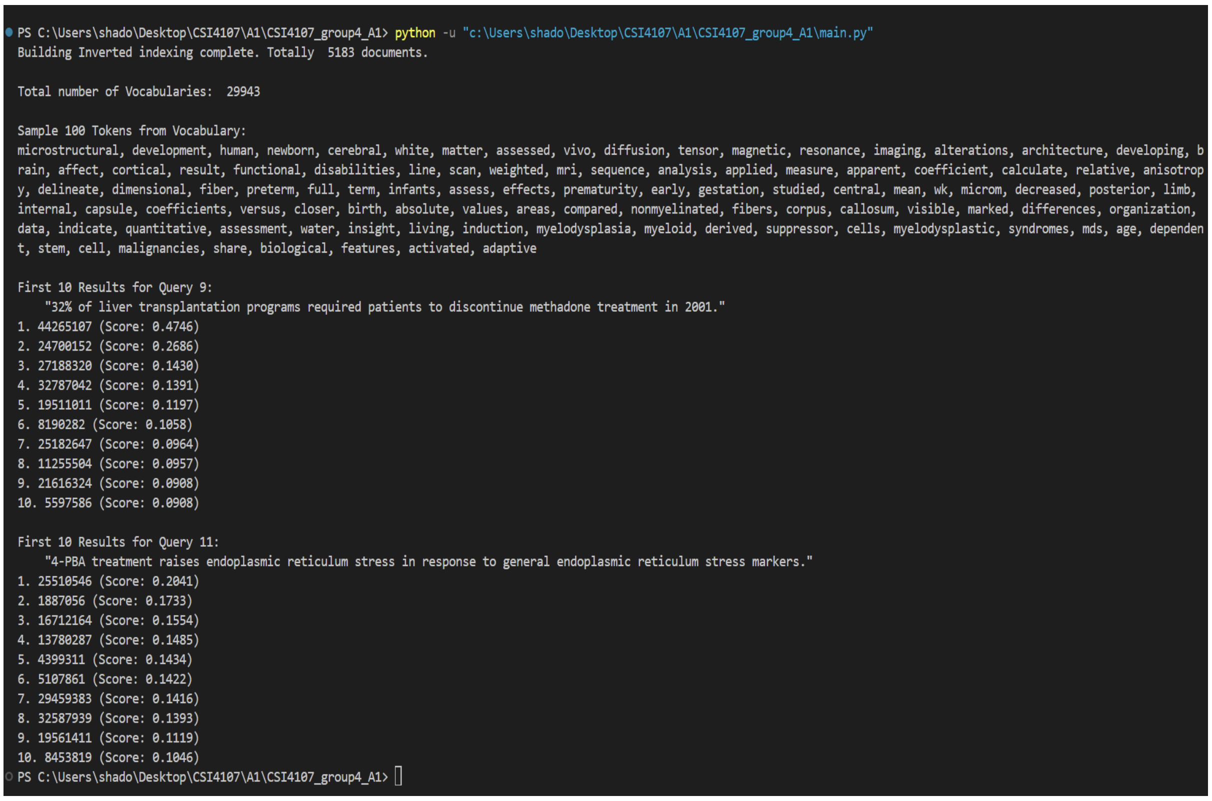
Execute the **main.py** file

- I.e. from a terminal window in the project folder: python3 main.py
  - Depends on python version installed on your machine
- May have to install dependencies for IRsystem.py
  - If imports in IRsystem.py are not recognized (i.e. you run main.py and it cannot recognize them) they can be imported using a python package manager (i.e. using pip: pip install <packageName>)
- Uncomments codes below the description for matching different requirements.

Algorithms, Data Structures, and Optimizations

For preprocessing we used simple regular expressions to filter out unwanted items. The main data structure used in our project is the inverted index, which was implemented using a python dictionary with the mapping of {Vocabulary Word (string), Set of Documents IDs for Documents Containing Word (set)}. Python dictionaries allow constant O(1) query time due to hashing, therefore querying for a specific vocabulary word is fast and consistent. We used a set object for the list of documents since we are primarily concerned with which documents are associated with each word, and querying for existence in a set is also constant O(1) query time due to hashing. This is helpful as during construction of the index, adding new documents to vocabulary words is fast and keeps our list of documents unique (since a set cannot contain duplicates). When it comes time to process a query, we can simply iterate over the query tokens and have a reasonably fast rank construction process. Finally, in our ranking function we use a standard library for cosine similarity.

See example results in Screenshot 1:



```
PS C:\Users\shado\Desktop\CSI4107\A1\CSI4107_group4_A1> python -u "c:\Users\shado\Desktop\CSI4107\A1\CSI4107_group4_A1\main.py"
Building Inverted indexing complete. Totally  5183 documents.

Total number of Vocabularies:  29943

Sample 100 Tokens from Vocabulary:
microstructural, development, human, newborn, cerebral, white, matter, assessed, vivo, diffusion, tensor, magnetic, resonance, imaging, alterations, architecture, developing, b
rain, affect, cortical, result, functional, disabilities, line, scan, weighted, mri, sequence, analysis, applied, measure, apparent, coefficient, calculate, relative, anisotrop
y, delineate, dimensional, fiber, preterm, full, term, infants, assess, effects, prematurity, early, gestation, studied, central, mean, wk, microm, decreased, posterior, limb,
internal, capsule, coefficients, versus, closer, birth, absolute, values, areas, compared, nonmyelinated, fibers, corpus, callosum, visible, marked, differences, organization,
data, indicate, quantitative, assessment, water, insight, living, induction, myelodysplasia, myeloid, derived, suppressor, cells, myelodysplastic, syndromes, mds, age, dependen
t, stem, cell, malignancies, share, biological, features, activated, adaptive

First 10 Results for Query 9:
    "32% of liver transplantation programs required patients to discontinue methadone treatment in 2001."
1. 44265107 (Score: 0.4746)
2. 24700152 (Score: 0.2686)
3. 27188320 (Score: 0.1430)
4. 32787042 (Score: 0.1391)
5. 19511011 (Score: 0.1197)
6. 8190282 (Score: 0.1058)
7. 25182647 (Score: 0.0964)
8. 11255504 (Score: 0.0957)
9. 21616324 (Score: 0.0908)
10. 5597586 (Score: 0.0908)

First 10 Results for Query 11:
    "4-PBA treatment raises endoplasmic reticulum stress in response to general endoplasmic reticulum stress markers."
1. 25510546 (Score: 0.2041)
2. 1887056 (Score: 0.1733)
3. 16712164 (Score: 0.1554)
4. 13780287 (Score: 0.1485)
5. 4399311 (Score: 0.1434)
6. 5107861 (Score: 0.1422)
7. 29459383 (Score: 0.1416)
8. 32587939 (Score: 0.1393)
9. 19561411 (Score: 0.1119)
10. 8453819 (Score: 0.1046)
PS C:\Users\shado\Desktop\CSI4107\A1\CSI4107_group4_A1> 
```

(Screenshot 1: Sample display showing size of vocabulary, 100 tokens from the vocabulary, and the first 10 results of first 2 queries)

Methods Evaluation

We have included the Mean Average Precision (MAP) scores in a file called MapScores.txt with the format Map | QueryID | Score

- Generated with the script (on macOS, UNIX system):
    - **./trec_eval-9.0.7/trec_eval -q -m map ./scifact/qrels/modified_test.tsv ./Results.txt > MapScores.txt**
        - We included trec_eval in our directory during testing
    - Note the original test.tsv did not contain the Q0 column so we build modified_test.tsv to add it (trec_eval was raising an error)
    - Those 2 comments should execute at the same time.

(For the Windows system, it requires installing the WSL to execute the above comments on the UNIX terminal.)

While scores for specific queries varied (see MapScores.txt) **our overall score (map, all, 0.5073) was 0.5073**, representing a moderate score where relevant documents are retrieved but there is still room for optimization.

We also ran the trec_eval script against our ResultsTitlesOnly.txt (query results using index built from document titles only) and the overall result was slightly worse (can check MapScoreTitlesOnly.txt). **The overall result was (map, all, 0.3610)** so building the index using the document text as well appears to be a more reliable method for relevance retrieval.

Summary

During this assignment we learned how to implement a basic retrieval system for a corpus of documents and tested it using a list of sample queries. Concepts learned/practiced include text processing, index building, relevance testing, and precision evaluation. While the information system built is relatively simple, it represents a solid foundation that can be used as a reference when advancing to more complex information systems.