CSI4107 Assignment 1 - Report

1. Team Information

- **Team Members:**
  - Ash Bhattarai (Student ID: 300236157)
  - Rudra Patel (Student ID: 300237682)
  - Georgin Binoy (Student ID: 300233721)
- **Task Distribution:**
  - **Preprocessing:** Ash developed preprocessor.py to clean and tokenize the corpus.
  - **Indexing:** Rudra was responsible for implementing indexing.py for building the inverted index and computing TF-IDF weights and document vector lengths.
  - **Retrieval & Ranking:** Georgin built retrieval.py to process queries, compute cosine similarity scores, and produce the final ranked list in TREC format.

Our IR system follows the standard vector space model with TF-IDF weighting and cosine similarity ranking. The implementation is divided into three core steps:

2.) Functionality

A. Preprocessing (preprocessor.py)

- **Input:**
  - Raw document file (corpus.jsonl)
  - Stopwords file (stopwords.txt)
- **Processing:**
  - Reads each JSON line from the corpus.
  - Combines available fields (e.g., title and text).
  - Removes URLs and extraneous Unicode characters.
  - Tokenizes using NLTK's TreebankWordTokenizer.
  - Converts tokens to lowercase, removes stopwords and punctuation, filters out tokens containing numbers, and applies stemming (via EnglishStemmer).
- **Output:**
  - A dictionary mapping each document ID to its list of processed tokens is saved as document_word_dict.json.
  - A separate frequency dictionary (document_word_count_dict) is optionally created.

B. Indexing (indexing.py)

- **Input:**
  - Preprocessed token dictionary (document_word_dict.json)
- **Processing:**
  - For each document, token frequencies are computed.
  - Constructs an inverted index where each token maps to:
    - A posting list (document IDs and raw term frequencies).
    - A document frequency (DF, i.e. number of documents containing the token).
  - Computes weights using the formula

$$w_{t,d} = tf_{t,d} \times \log\left(\frac{N}{df_t}\right)$$

    Computes the Euclidean length of each document vector (to later normalize cosine similarities).

- **Output:**
  - The weighted inverted index is saved as inverted_index.json (or as weighted_dict.json if renamed).
  - Document vector lengths are saved as document_vector_lengths.json.

C. Retrieval & Ranking (retrieval.py)

- **Input:**
  - Weighted inverted index (loaded from weighted_dict.json)
  - Preprocessed queries from queries.jsonl
  - Stopwords (same as used in preprocessing)
- **Processing:**
  - Each query is preprocessed using a similar pipeline as the corpus.
  - Constructs a query vector with a variant of TF-IDF weighting. In our implementation, we use the formula:

$$w_{q,t} = \left(0.5 + 0.5 \times \frac{tf_{q,t}}{\max(tf_q)}\right) \times idf_t$$

  - For each query, documents containing at least one query term are identified from the inverted index.
  - Computes the cosine similarity

$$\cos(q,d) = \frac{\sum_{t \in q \cap d} w_{q,t} \times w_{t,d}}{\|q\| \times \|d\|}$$

using precomputed document norms.

  - Ranks the documents by descending score.
- **Output:**
  - The final retrieval results are saved in results.txt.
  - Processed queries are also saved to query_processed.json for reference.

3.) How to Run the program

Prerequisites

Before running the program, ensure that you have the following installed on your system:

- Python 3.x
- Required Python libraries (NLTK, NumPy, Scikit-learn, Pandas, json)
- trec_eval (for evaluation)

Running the System

- Navigate to the project directory:
  cd ~/Downloads/CSI4107_A1-main
- Preprocess the dataset:
  python3 preprocessor.py
- Build the inverted index:
  python3 indexing.py
- Perform retrieval and ranking:
  python3 retrieval.py
- Run the evaluation using trec_eval:
  trec_eval -m map scifact/qrels/test_fixed.tsv results.txt

Expected Output

- If successful, trec_eval will output the Mean Average Precision (MAP) score in the format:
  map all 0.4348

Troubleshooting

- If results.txt or test.tsv is missing, ensure that files are correctly generated in the expected directories.

- If trec_eval reports a formatting issue, check that test.tsv follows the required TREC qrels format:
  query-id 0 doc-id relevance-score

Example:
1 0 31715818 1

4.) Additional Analysis

Algorithms, Data Structures, and Optimizations in Each Step

Step 1: Preprocessing (preprocessor.py)

Algorithm Used:

- Tokenization using TreebankWordTokenizer
- Stopword removal based on a predefined stopwords list
- Stemming using EnglishStemmer
- Filtering of tokens:
  - Removing URLs
  - Removing punctuation
  - Removing numbers
  - Removing certain unwanted Unicode characters

Data Structures:

- **Dictionary (document_word_dict)**: Maps each document ID to a list of preprocessed words.
- **Set (stopwords)**: Efficient look-up for stopword filtering.
- **Counter (document_word_count_dict)**: Stores word frequency in each document.

Optimizations:

- Used **set lookups** for stopwords removal (O(1) complexity per word).
- Applied **stemming** to reduce vocabulary size and improve generalization.
- Removed **unnecessary tokens** (e.g., punctuation, numbers, and stopwords) early to speed up indexing.

Step 2: Indexing (indexing.py)

Algorithm Used:

- Construct **Inverted Index** (token → {doc_id → term frequency})
- Compute **TF-IDF weights**:
  - IDF=log(Total DocumentsDF)IDF=log(DFTotal Documents)
  - Weight=TF×IDFWeight=TF×IDF
- Compute **document vector lengths** for cosine similarity retrieval.

Data Structures:

- **Dictionary (inverted_index)**:
  - Key: Token
  - Value: { "postings": {doc_id: TF}, "df": document frequency }
- **Dictionary (doc_vector_lengths)**: Stores the Euclidean norm for each document.

Optimizations:

- Used a **single pass dictionary update** to avoid redundant token processing.
- Stored **document frequencies (DF)** alongside postings to avoid recomputation.
- Used **logarithmic scaling** for IDF to prevent large differences in term importance.

Step 3: Retrieval (retrieval.py)

Algorithm Used:

- Preprocess the query text similarly to documents.
- Compute **query vector** using:
  - Weight=(0.5+0.5×TFmaxTF)×IDFWeight=(0.5+0.5×maxTFTF)×IDF
- Perform **Cosine Similarity Ranking**:
  - Compute dot product of query and document vectors.
  - Normalize scores using Euclidean norms.

Data Structures:

- **Dictionary (weighted_dict)**: Maps each token to document TF-IDF weights.
- **Dictionary (doc_norms)**: Precomputed document vector norms for efficient ranking.
- **Dictionary (query_vector)**: Query term weights.

Optimizations:

- Used **precomputed document norms** to speed up similarity calculations.
- **Skipped missing query terms** to reduce computational overhead.
- Stored **TF-IDF in a compressed JSON format** to optimize memory usage.

Step 4: Vocabulary Analysis:

Sample 100 tokens from vocabulary: ['cyclobutan', 'anthropomorph', 'obliqu', 'ifngammainduc', 'actinexpress', 'gavag', 'scarc', 'wholeorgan', 'autophagy', 'phenolsulfotransferas','commensur', 'cd', 'unrestrict', 'cellsort', 'mossi', 'blockag', 'spiral', 'entries', 'hf', 'nucleotidedepend', 'pericentr', 'epstein', 'stapf', 'glossari', 'bistability', 'anillin', 'nadphrequir', 'gproteincoupl', 'superimpos', 'deep', 'tf', 'impli', 'thromboxan', 'profibrogen', 'swro', 'unpolymer', 'follicular', 'immunitydiseas', 'labeling', 'ifa', 'coincident', 'took', 'tle', 'leucocytosi', 'hn', 'desorb', 'electrophil', 'overlooked', 'lncrnacarel', 'cellosteoclast', 'thrombomodulin', 'herit', 'sxs', 'generat', 'evening', 'eia', 'scholz', 'higherlevel', 'bidimension', 'activecontrol', 'bronchiol', 'multipledos', 'modelderiv', 'endprocess', 'applications', 'spb', 'reseal', 'easi', 'doxorubicininduc', 'diseas', 'puritan', 'feedforward', 'storage', 'lpbn', 'waveforms', 'piwi', 'uroporphyrinogen', 'pdoxgmtinduc', 'assembl','vti','hematocrit', 'histologyindepend', 'ntb', 'bbs', 'strait', 'haematogen', 'flowdepend', 'antioxid', 'emergence', 'ablat', 'surgeri', 'fpg', 'genesoligonucleotid', 'largdeplet', 'creb', 'fih', 'forapoegenotyp', 'mucin', 'prophase', 'generaliz']

Step 5: First 10 Answers for the First 2 Queries

Query 1:

Rank	**Document ID**	**Score**
1	13231899	0.3527
2	10906636	0.0724
3	994800	0.0718
4	21439640	0.0651
5	21257564	0.0596
6	20490533	0.0593
7	15115749	0.0590
8	19177164	0.0583
9	18757553	0.0579
10	24617927	0.0575

Query 2:

Rank	**Document ID**	**Score**
1	15331584	0.4328
2	18050287	0.1212

3	12068388	0.1014
4	19182325	0.0983
5	23174355	0.0959
6	3720107	0.0638
7	12650610	0.0628
8	6673421	0.0624
9	23294314	0.0610
10	2543135	0.0595

Discussion of Results

- **Title vs. Full Query:** Using **title + full text** in queries generally improves results compared to just titles.
- **Ranking Performance:** The first document in **Query 1** has a much higher score than others, suggesting a strong match.
- **Effectiveness:** The TF-IDF model performs well but may need further tuning (e.g., BM25 weighting or query expansion).

Step 6: Mean Average Precision (MAP) Score

To evaluate the performance of our Information Retrieval (IR) system, we used **trec_eval** to compute the **Mean Average Precision (MAP)** score on the **test queries**.

The obtained **MAP score is 0.4348**, which indicates the system's ability to retrieve relevant documents ranked high in the results.

Analysis of Retrieval Performance

- The MAP score of 0.4348 suggests that the system retrieves relevant scientific abstracts with moderate accuracy.
- The retrieval performance was evaluated using TF-IDF weighting, and an alternative run was tested with BM25. BM25 yielded higher MAP scores, demonstrating its advantage in ranking relevant documents.
- Future improvements could include pseudo-relevance feedback and query expansion techniques to enhance retrieval effectiveness.