Instructions

You are tasked with building a search engine that efficiently stores and retrieves terms from a small corpus of documents. Follow the steps below to complete the task.

Questions

1. **Dictionary and Posting Compression**

   (a) **Preprocess the Documents**

   You are given a set of documents as lists of words (terms). Your first task is to preprocess the documents by:

   - Converting all terms to lowercase to ensure case-insensitivity.
   - Removing any punctuation.
   - Sorting the terms in alphabetical order.
   - Creating unique term lists for each document, to be used in dictionary and postings list generation.

   (b) **Create Dictionary and Postings Lists**

   i. **Dictionary**: Extract unique terms across all documents to form a dictionary, and sort the terms alphabetically.

   ii. **Postings Lists**: Create a postings list for each term in the dictionary. Each postings list contains the **document IDs** (starting from 1) of documents where the term appears. Ensure that the document IDs in each list are sorted in ascending order.

   (c) **Apply Front-Coding to Compress the Dictionary**

   To compress the dictionary, use **front-coding** to store terms with shared prefixes efficiently:

   i. **Divide** the dictionary into blocks of four terms each.

   ii. **Store** the first term of each block in full. For each subsequent term in the block, store only the prefix length (number of shared characters with the previous term) and the unique suffix.

   Write a function to generate the compressed dictionary using this format.

(d) **Apply Gamma Encoding to Compress Postings Lists**
For each term in the dictionary, the postings list should be compressed as follows:

i. Apply **delta encoding** to the postings list by storing the differences (or gaps) between consecutive document IDs.
ii. Apply **gamma encoding** to each delta.

Write a function to generate the compressed postings lists.

(e) **Answer Queries Based on Compressed Dictionary and Postings Lists**
Once your dictionary and postings lists are compressed, implement a query system that allows you to answer boolean queries.

2. **Vector-Space Model**

(a) **Use Preprocessed Dataset from Previous Part**
Begin with the preprocessed dataset provided in the last assignment.

(b) **Create TF-IDF Matrix**
Compute the **TF-IDF matrix** for the dataset.

(c) **Implement Vector-Space Model**
Implement the **vector-space model** to represent each document as a vector of TF-IDF weights.

(d) **Retrieve Queries with and without Weight Normalization**
For each query, retrieve the top-ranked documents(k=4) based on **cosine similarity**. Perform retrieval twice:

- **With weight normalization**
  Normalize document vectors before calculating cosine similarity.
- **Without weight normalization**
  Use the raw TF-IDF values in the document vectors for similarity calculation.

(e) **Visualize Retrieved Documents based on Cosine Similarities**
Plot the retrieved documents in a 2D space based on their **cosine similarities** to the query. Label the query and the top retrieved documents in the plot to show their relative similarity.

- How does normalization impact the retrieval results? Discuss any differences observed in the top-ranked documents between the normalized and non-normalized methods.
- Based on the visualization, what insights can you draw about the distribution of documents in vector space?
- Suggest one or two ways to further enhance the retrieval model based on your findings.

3. **Term-At-A-Time Processing**

(a) **Use the TF-IDF Matrix**
- Utilize the TF-IDF matrix created in the previous part.

(b) **Implement Pivot Normalization**
- Apply pivot normalization to adjust the TF-IDF values in the matrix.

(c) **Implement Term-at-a-Time Processing**
- Develop a function that processes each term in the query individually, calculating the relevance scores for each document based on the normalized TF-IDF values.

(d) **Retrieve Documents Using Mean-Heap**
- Retrieve the documents based on the given queries using term-at-a-time processing. Utilize a min-heap to maintain the top k=4 documents with the highest relevance scores during retrieval.

- Why is pivot-normalization used in the context of document retrieval?
- What steps do you take to choose an appropriate pivot value, and how does this choice affect the normalization process?
- How Does term-at-a-time-processing differ from document-at-a-time processing, and what are the advantages and disadvantages of using term-at-a-time processing in information retrieval?
- Discuss the role of a min-heap in the ranking process during document retrieval. How does using a min-heap enhance the efficiency of retrieving the top k documents?

Submission:
Please submit your answers to the Quera containing .ipynb file for your codes, .py file, and a single PDF file for codeless answerings.

References:
Lectures 4-6 of Advanced Information Retrieval course instructed by Dr. Bagher BabaAli, Fall 2024, University of Tehran.