Data Structures and Algorithm

COL106

Assignment 7

QnA Tool with Search Engine

Prateek Maurya 2022MT11937

Arnav Raj 2022CS51652

Suhani Soni 2022MT61981

Akarsh Gupta 2022MT61966

Aim

1. In this assignment the key task to perform is the query function.
2. We are working with LLMs here and we are given a large corpus from which we are meant to seek answers for random questions. We implemented a scoring mechanism which helps us determine the importance of a word or a paragraph with respect to the question asked.
3. We ultimately need to get the top K relevant paragraphs.
4. The assignment also involves the optimization of the query function which utilizes ChatGPT API.

Algorithm

1. We have applied the Term Frequency – Inverse Document Frequency algorithm here. Term Frequency, hereby referred to as TF, measures how frequency a term occurs in the paragraph. It has two input parameters : the term, and the paragraph.
2. $TF = ln$(Number of times the term appears in Paragraph/ Total Number of Terms in Paragraph) + 1.

3. On the other hand we have the Inverse Document Frequency, which measures how frequently a term appears in ALL paragraphs. Notice the distinction made here : TF measures frequency of a particular term for a particular paragraph, while IDF measures frequency of appearance for a particular term.

4. $IDF = ln$(Total Number of Paragraphs/Total Number of Paragraphs containing the term) ($IDF = 0$ if term does not appear)

5. Some explanation may be in order : suppose a simple word "is" ; naturally this will occur in multiple paragraphs : almost all the paragraphs. It is clear mathematically that IDF then will be close to 0. By using IDF we reduce the impact of such commonly occurring words in search results.

6. We combine these attributes to finally get a relevance score : *Relevance* (Term, Paragraph) = TF(Term, Paragraph) * IDF (Term)

Algorithm in Our Code

1. We initially conduct a word search. The function *wordResults*(pattern) is responsible for returning a vector of $EndWord*$ objects, each representing a word that matches the provided search pattern. These words serve as the basis for identifying relevant paragraphs in subsequent stages.

2. Then we create an instance of the *ParagOccur* tree is created. This tree structure is designed to store the occurrences of words in paragraphs, for efficient retrieval of this data.

3. The code then traverses the occurrences of each word obtained from the search pattern and inserts into the *ParagOccur* tree. For each paragraph where a word is found, the tree is updated to reflect that occurrence.

4. The code moves on to the calculation of the query vector. This vector represents the search query in a mathematical form (dot product) and is used for later stages of the search algorithm. For each word in the input query, the code calculates the Inverse Document Frequency (IDF) value, which reflects the *rarity* of the word across the entire document corpus. These IDF values are then used to construct the query vector

5. The normalization of the query vector, which we perform by computing its norm, for which we have defined a *helper* function, ensures that the vector is appropriately scaled for subsequent operations. This helps prevent any biases in search operations (*is* might occur multiple times in different paragraphs and might cause a problem)

6. With the query vector prepared, the search engine proceeds to identify relevant paragraphs. The tree is traversed again, and for each paragraph, a dot product is calculated. This dot product operation involves combining the TF-IDF values of the words in the search pattern with the occurrences of those words in the paragraph. The result is a score representing the relevance of each paragraph to the search query. This score finally gives a sort of *ranking* to each element within the search engine

7. Following the computation of paragraph scores, the code enters a normalization phase. We make sure that the scores are normalized by dividing each by the product of the norm of the paragraph vector and the norm of the query vector. This normalization ensures that the scores are *comparable* across different paragraphs, considering their varying lengths and the characteristics of the search query.

8. The final step involves sorting the paragraphs based on their scores, presenting the results in descending order of relevance. For this we defined a *helper* function named *sort* which is nothing but Merge Sort , where the sorting takes place on the basis of the score. The sorted vector of 'ParagOccur::Node*' is then returned as the output of the search operation.

Stop Words Mechanism

1. Stop Words and Hash Table Implementation : We define two classes: *stopwordsnode* and *stopwordshashtable*. The *stopwordsnode* class represents a node and holds a single string, which is a stop word. The *stopwordshashtable* class is designed to manage an array-based hash table, where each element of the array is a vector of *stopwordsnode*. This structure is used to efficiently store and retrieve stop words.The hash function used for indexing into the table takes a string (presumably a stop word) as input and applies bitwise XOR and multiplication operations to generate an index. This index is then used to place the stop word in the corresponding vector within the hash table.

2. Stop Words Insertion :The *stopwordshashtable* class includes a method *insertstopwords*() for populating the hash table with common stop words. It initializes an array of stop words and, for each word, calculates its hash using the hash function and inserts it into the appropriate vector in the hash table. We have performed linear probing here.

3. Stop Words Lookup : The *isStopword* method in the *stopwordshashtable* class is responsible for checking whether a given word is a stop word. It does this by computing the hash of the input word and then searching for a matching entry in the corresponding vector. If a match is found, it returns 'true', indicating that the word is a stop word; otherwise, it returns 'false'.

4. Query Modification : The $ModifyQuery$ method takes an input query string and processes it to filter out stop words. It uses an instance of the *stopwordshashtable* class to perform the filtering. It iterates over each character in the input query, identifies words based on separators provided by the *separator* method.

   For each identified word, the method checks whether it is a stop word using the *isStopword* function from the hash table. If the word is not a stop word, it is appended to the $ModifyQuery$ string. Separators are appended directly *without* any modification.

   This is to ensure that the stop words are filtered out from the query, and the modified query is constructed without these common terms.

Helper Functions and Comments

1. Insert Helper : For the dictionary we have this simple insert function which takes in a node containing a word and adds it according to its weight ; it is taken care to make sure that the tree is balanced by considering cases for various kinds of rotations and height adjustments.

2. Get Paragraph : Given details like page number etc. , we search for the relevant paragraph by comparing the meta data of paragraphs with given input. If it matches, we have our paragraph.

3. Feed Dictionary : This helper function takes in a document which contains sentences and paragraphs and considers words from it ; it analyzes the elements from it to understand if the element is a word or number, it consequently adds the sentence to the dictionary element by element by the tools we have defined for the dictionary.

4. Merge Sort for Search Engine : We sort different words on the basis of their counts by using Merge Sort

5. Word Results : We process the words and add them into a vector of EndWords pointers while taking care of separators etc.

6. Paragraph Search Results : We use the word results function to process the words and we subsequently add them into trees ; we then use our merge sort function to sort paragraphs on the basis of their scores. The making of linked list is done separately since the relevant streams are not available here.

7. Get Top K Para : Here we use they paragraph search results function to process the query and then get relevant paragraphs ; we keep a counter of number of paragraphs we have resulted into the output and make sure it does not exceed K.

8. Other functions : Like that of scoring, checking the presence of separator or some functions of tries like rotations etc. are also made here

9. We added the following "The information provided below are excerpts from books and letters written by Mahatma Gandhi in first person. You are a knowledgeable and helpful assistant. You read the text given below and answer queries based on the text. Write an detailed answer to the question at the end of this text using the information in the text, you can also put in extra pieces of information as you deem fit in relevance to the question :" in *QueryLLM* function to be included in *query* txt file for better answers to query based on paragraphs retrieved.

10. We used the cmath for using the logarithm function