# COL106 ASSIGNMENT 7
# Corpus Q and A Tool

Date: 14-11-2023
Course: COL106

Report submitted by:
Sneha Pareek(2022CS11096)
Aastha A K Verma (2022CS11607)
Pradyuman Singh (2022CS11122)
Anamika(2022CS11098)

## 1. Aim

In the part 2 of the assignment: QUERING THE LLM asked to extract the most pertinent data from the corpus in response to a query that would be fed into the large language model.

## 2. Algorithm

Algorithm includes several steps as mentioned below:

- Storing the unique words in the corpus
- Query Preprocessing
- Ranking the Paragraphs
- Feeding extracted data and query to LLM

### 2.1 STORING THE UNIQUE WORDS IN THE CORPUS

#### DATA STRUCTURE USED AND WHY THIS DATA STRUCTURE

- We used trie data structure to store the words along with their frequency in the corpus. At the end of each word that is inserted in the trie there is a linked list in which the information book code, page no, paragraph no of each occurrence of the word in the corpus is stored in the nodes. Also, the length of linked list so formed is also stored at the end of the word so that to whenever the frequency of a word in the corpus is needed, we do not have to iterate through the complete linked list.
- Insertion as well as search of a word in the trie is faster, it only takes O(length of word) for both of these operations.

## 2.2 QUERY PREPROCESSING

### WHY QUERY PREPROCESSING?

- Query preprocessing is necessary to improve the efficiency of information retrieval from the corpus based on the query provided. It ensures that the search process is more accurate, efficient, and capable of understanding the user's intent by transforming the raw query into a form that is compatible with the structure and content of the corpus.
- We first tokenized the query provided, keeping only words, removing punctuation marks and stored the tokens in a vector for the purpose of further processing.

### TECHNIQUES USED FOR PREPROCESSING:

- **Normalisation:** Normalizing the case of all characters in the query (e.g., converting everything to lowercase) ensures case insensitivity in the search. This prevents mismatches that could occur if the case of the query does not match the case of the text in the corpus. We too converted the complete query in lower case before doing any other processing considering this reason.
- **Stopword Removal:** Remove common words (stopwords) that do not contribute much to the meaning of the query. Examples include "and," "the," "is," etc. This reduces noise and focuses the search on more meaningful terms. We included a set of stop words and if any token in the tokenized query matched with any of them then that word was ignored. Then the frequency of the left relevant words is the query is stored along with word. This frequency of each word is known from trie in which all the words were stored.
- **Part-of-Speech Tagging:** By taking into account the significance of particular parts of speech in the query, POS tagging can help improve the ranking of search results. An appropriate ranking of pertinent texts can be achieved by giving weight to terms according to their grammatical type. In our algorithm we used POS tagging and then added a marker to the end of proper nouns so that later on in the algorithm they can be given more weight. This aids in more precisely retrieving documents.
- **Stemming and Lemmatization:** Reduce words to their root or base form. For example, "running" and "ran" may be stemmed to "run." This helps in capturing the core meaning of words and increases the recall of relevant documents. It promotes consistency in document retrieval by treating morphological variations as the same word. This helps in eliminating redundancy and ensures that relevant documents are not overlooked due to variations in word forms.
- **Handling Synonyms:** Queries may contain synonyms or semantically related terms. Users may express the same concept using different words or phrases, and by expanding the query to include these variations, the system is more likely to capture a comprehensive set of relevant documents. Relying solely

on exact word matches may lead to missed opportunities in information retrieval. Including synonyms allows the system to be more forgiving of slight variations in the way users express their information needs, reducing the dependency on exact word matches. To give more important to some words we added different number of synonyms of different words. The number of synonyms of a word considered in the list of the words to be searched in the corpus are inversely proportional to the frequency of the word in the corpus. The more frequent a word is in the corpus less of its synonyms are taken into consideration. Also, all the synonyms of a word have not been given equal importance.

- **Weighting and Ranking:** Assigning weights allows the system to prioritize terms that are more likely to be central to the user's query. Some parts of speech may contribute more to recall (e.g., nouns for broad concepts), while others may enhance precision (e.g., adjectives for specificity). Weights help strike a balance between the two. Initially, each of the relevant word left in the query were given weights inversely proportional to their frequency in the corpus. Then, the weight of the proper nouns was increased by a factor of $e^{1.5}$. In the list of synonyms of a word the weight of first synonym has been increased by a factor of $e^{1}$ and for the second by $e^{0.9}$ and in this way the multiplication factor is being decreased by $e^{0.1}$ with each next synonym.
- Finally, all of this processing we get a vector of vector of string in which the first element is the token, second element is a string that lets us know whether the token is a proper noun or not and the third element is the weight of that token.

## 2.3 RANKING THE PARAGRAPHS

For ranking the paragraphs, we have followed the following steps:

- For the purpose of keeping the paragraphs along with their score at any stage of the program we have made a vector of vector of vector of long double so that we can access a paragraph in a particular book at a particular page in O(1) just by indexing.
- The vector of strings containing tokens to be considered for ranking of the paragraphs is passed as an argument to the **BMScore** function along with the trie containing all the words of the corpus.
- Then corresponding to each token, the score of the paragraphs in which that word has occurred is updated using the BM algorithm and then the para nodes of the paragraphs whose scores have been updated are inserted in a new vector **to_be_sorted**.
- Finally, when all the tokens have been iterated and used to update the score of the paragraphs. All the paranodes that are present in the vector **to_be_sorted** are used to make **master_para_nodes** that along with the information of the paragraphs that was present in the para nodes also

contains the score of each paragraph. Then all of these **master_para_nodes** were inserted in **to_be_inserted** vector.

- A visited vector too was maintained so that once a node has been inserted in to_be_sorted it is marked as visited and when again the score of the same paragraph was updated we need not push the same para node to the vector again but access it from the **to_be_sorted** vector directly.
- Then we used min heap to extract k top scored paragraphs. We build a min heap of k nodes and then compared the rest of the master_para_nodes in the to_be_inserted vector with the root and if the score of the root was less then the node with which it was being compared then the root was replaced with the new node and heapified down.
- At last, the heap had top k scored **master_para_nodes** in it. Then this final heap is sorted using heap sort and a linked list of para nodes of these top k scored paragraphs is made and the pointer to the last node of the linked list is returned by the **BMScore** function.
- By the member function get_top_para2 of **QNA_tool** this **BMScore** function is returned.
- This linked list containing the information of the top k paragraphs corresponding to a query is feed to LLM along with the question to be answered in the **query_llm** function.
- LLM then based on the paragraphs provided and the query being asked generated a response. Which feeding these paragraphs to LLM we have given command not to use information other than provided by the paragraphs to answer the query.