COL106

ASSIGNMENT-7 Corpus Q&A Tool

REPORT

Contributions by:

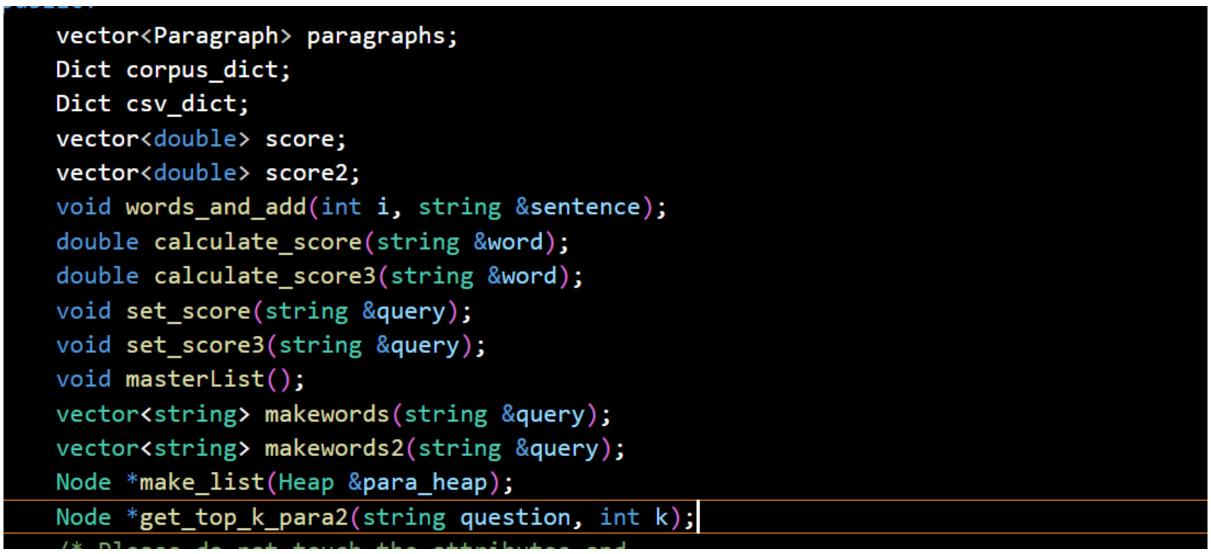
Dhruv Belawat	**Toshi Pahadia**	**Shikhar Gupta**	**Rohan Roy**
2022EE11661	2022CS11127	2022MT11925	2022MT11294

AIM

In this project, we were given creative liberty in selecting algorithms and data structures to enhance the performance. The task involves the optimization of a query function, denoted as query_llm, which utilizes the ChatGPT API.

The main crux of this assignment lies in this function called query. This function uses many helper functions such as get_top_k_para2 which is a modified version of get_top_k_para used in part 1.

HELPER FUNCTIONS



```
    vector<Paragraph> paragraphs;
    Dict corpus_dict;
    Dict csv_dict;
    vector<double> score;
    vector<double> score2;
    void words_and_add(int i, string &sentence);
    double calculate_score(string &word);
    double calculate_score3(string &word);
    void set_score(string &query);
    void set_score3(string &query);
    void masterList();
    vector<string> makewords(string &query);
    vector<string> makewords2(string &query);
    Node *make_list(Heap &para_heap);
    Node *get_top_k_para2(string question, int k);
```

This function has several other helper functions defined in it. Let’s have a look at them one-by-one:

1) **QNA_tool Class:** Implements various functionalities such as stemming words, calculating word scores, and employing a heap data structure for paragraph selection based on relevance. Defines methods for obtaining the top k relevant paragraphs based on a given query, considering factors like word scores. It includes methods such as the constructor (QNA_tool()), destructor (~QNA_tool()), masterList(), insert_sentence(), words_and_add(), calculate_score(), calculate_score3(), make_list(), makewords(), makewords2(), get_top_k_para(), get_top_k_para2(), set_score(), set_score3(), query(), and get_paragraph().

2) **Heap Class:** This class appears to implement a heap data structure, including methods like heapup (), heapdown (), removetop (), etc.The heap is likely used to prioritize and efficiently retrieve top-scoring paragraphs.

3) **Make_words2:** It takes the question or query and returns a vector containing all the keywords in it. This is done by splitting the string question and then omitting all the irrelevant words like prepositions and articles like “a”, ”an”, ”the” because these words occur very frequently and irrelevant paragraphs containing a lot of these words will be selected if these words aren’t omitted. We also removed common words like “which”,”why”,”where” which might appear in queries but are irrelevant.

1) **Stemmer:** This is a very important function based upon the **Porter-Stemmer Algorithm.** The basic job of this function is to implement a small scale version of the porter stemmer algorithm wherein suffixes are removed from the word. Ex- “cats” are converted to “cat” and “sleeping” is converted to “sleep”. The actual porter stemmer algorithm has a variety of rules and a very complex algorithm is followed, but this function is a condensed version of the same. **Note: We are using both the word and the stemmed word for calculating the score of the relevant paragraphs**. Ex: both cat and cats will be used for searching for the relevant paragraphs. This function takes a word as input and attempts to remove common word suffixes like "ing," "ed," "ly," "es," and "s."

2) **Calculate_score3 ():** Evaluates the score for the word using the corpus count and csv count. Since the term for csv count is in the denominator it implies that more frequent words will have a higher csv count and hence a lower score. This will make less frequent words (which are mostly keywords) dominating in terms of score.

- Input: The function takes a string `word` as input. The function retrieves the frequency of the word in the corpus (`corpus_count`) and the frequency from the CSV file (`csv_count`).
- Score Calculation: The score is calculated using the formula:
    - score = 1000000 / ((1 + corpus_count) * (1 + csv_count))

  The addition of 1 in the denominators avoids division by zero and biases the score towards less frequent words.
- Stemming Check: The function checks if stemming the word results in a different word (`query2`). If so, an additional score is calculated for the stemmed word using the same formula. This step aims to capture variations of the word, enhancing the scoring mechanism.

- An important point to note here is that words that frequently occur in the corpus are also generally irrelevant. So the corpus count term has also been included in the denominator as compared to part 1 of the assignment wherein it was in the numerator.
- It would have been preferred that in the numerator instead of frequency, log(frequency) of the word would have been used because when too many words are matching, the percentage difference in the importance of the paragraph is reduced.
- The final score is returned.

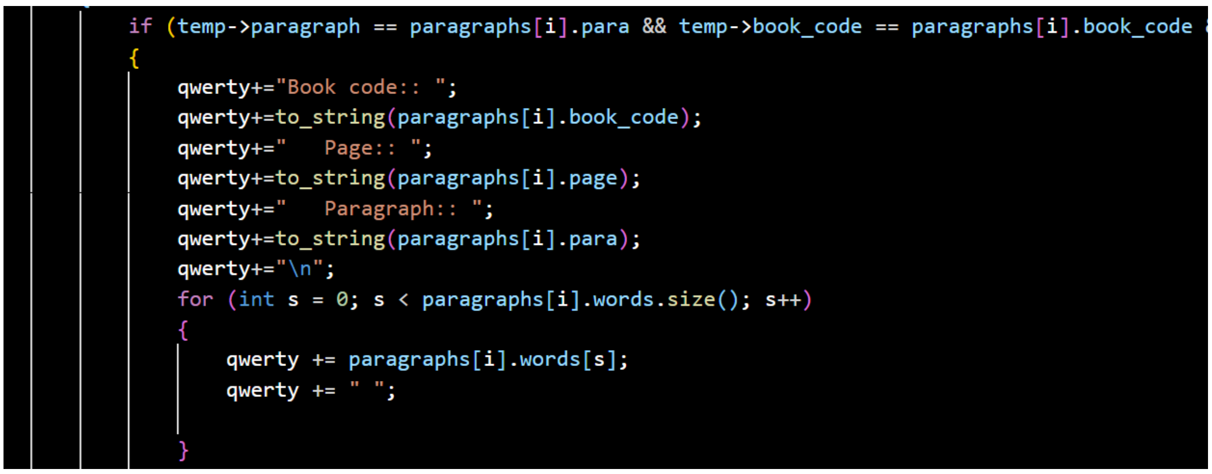
3) **Make_list:** This function converts the given heap into a linked list.

4) **File Reading and Processing:** The masterList () method reads data from a CSV file (unigram_freq.csv) and populates a data structure (csv_dict) with word frequencies.The insert_sentence () method adds sentences to paragraphs, and the words_and_add () method processes words within a sentence.

5) **Get_top_k_para2 ():** This function is responsible for retrieving the top-k paragraphs based on a given question. The paragraphs are ranked using a scoring mechanism implemented in `calculate_score3 () `. The top-k paragraphs are then organized into a linked list, and a reference to the head of the list is returned. - The function takes a string `question` and an integer `k` as input. The question is tokenized into words using the `makewords2 () ` function. This function removes common English stop words (e.g., "the," "of," "and") and stems the remaining words.

- Score Calculation: The `set_score3 () ` function is called for each word in the tokenized question. The `set_score3 () ` function computes a score for each paragraph based on the frequency of the word in the paragraph and a scoring mechanism implemented in `calculate_score3 () `.
- Heap Construction: A heap (`paraheap`) is constructed to store the top-k paragraphs based on their scores. The top-k paragraphs are identified and added to the heap. The heap maintains the paragraphs with the highest scores.
- A linked list (`Node` structure) is created from the paragraphs in the heap.

6) **Query Handling:**
The query () method processes user queries, identifying top paragraphs based on the query and the scoring mechanism. The query_llm () method writes relevant information to files, executes a Python script, and manages the interaction with a language model (LLM) using an API key.
The query has been written in absolute clear details so that the LLM which in our case in chatgpt can receive instructions properly.
The query fed to the LLM includes the bookcode, page number and paragraph number just to better show that the context is taken from an extract.



```
if (temp->paragraph == paragraphs[i].para && temp->book_code == paragraphs[i].book_code
{
    qwerty+="Book code:: ";
    qwerty+=to_string(paragraphs[i].book_code);
    qwerty+="    Page:: ";
    qwerty+=to_string(paragraphs[i].page);
    qwerty+="    Paragraph:: ";
    qwerty+=to_string(paragraphs[i].para);
    qwerty+="\n";
    for (int s = 0; s < paragraphs[i].words.size(); s++)
    {
        qwerty += paragraphs[i].words[s];
        qwerty += " ";

    }
```

Then the following string is added at the end of the query : `"With respect to the above given context, answer the below question in multiple paragraphs the best possible way and give the necessary references in points."`

The above string is of high importance. Firstly we are requesting for multiple paragraphs in the answer. This ensures that small answers and one liners aren't given in the answer. Further necessary references are asked for which ensures that the answers are detailed and provided with information regarding where they are extracted from.