The Algorithm Used to feed the appropriate text to LLM:

1. Word Scoring (s(w)):

For each word **w**:

s(w)=log(frequency(w) in specific corpus+1) – log(frequency(w) in general corpus+1)

2. Term Frequency (TF) for a paragraph:

For each word w in the paragraph **p**:

tf(w,p) = Number of occurrences of w in p / Total No. of words in p.

3. Inverse Document Frequency (IDF) for a Word across Paragraphs:

Idf(w) = log(Total number of unique paragraphs in the corpus) – log(No. of paras containing w)

4. TF-IDF for Paragraph Scoring:

For each word w in the paragraph **p**:

tf-idf(w, p) = tf(w , p) x idf(w)

5. Score of a Paragraph (score(p)):

For the paragraph p containing words w_1 , w_2, …… w_k:

Score(p) = ∑ tf-idf(w_i , p) x s(w_i)

6. Normalization of Paragraph Scores:

Normalize the paragraph scores:

Normalized score(p) = score(p) / total number of words in p

Code Snippet for the Paragraph scoring Algo used:

```
Node* QNA_tool::get_top_k_para_ultra(string question, int k) {
    question.push_back(' ');
    vector<double> scores(data.size() , 0.0);
    vector<string> query_words = get_words(question);
    double s , idf , tf , tfidf;
    for(auto query_word : query_words) {
        s = log(static_cast<double>(inserted_corpus -> get_word_count(query_word) + 1) / static_cast<double>(general_corpus -> get_val(query_word) + 1));
        idf = log(static_cast<double>(data.size() + 1) / static_cast<double>(no_of_para(query_word , data) + 1));
        for(int i = 0 ; i < (int)data.size() ; i++) {
            vector<vector<string>> paragraph = data[i];
            int occurrences_in_para = 0;
            int words_in_para = 0;
            for(auto sentence : paragraph) {
                words_in_para += sentence.size();
                for(auto word : sentence) {
                    if(word == query_word) {
                        occurrences_in_para++;
                    }
                }
            }
            tf = static_cast<double>(occurrences_in_para + 1) / static_cast<double>(words_in_para + 1);
            tfidf = tf * idf;
            scores[i] += tfidf * s;
        }
    }

    for(int i = 0 ; i < (int)data.size() ; i++) {
        vector<vector<string>> paragraph = data[i];
        int words_in_p = 0;
        for(auto sentence : paragraph) {
            words_in_p += sentence.size();
        }
        scores[i] = scores[i] / static_cast<double>(words_in_p);
    }
```

```
  Node* sentinel = new Node(-1 , -1 , -1 , -1 , -1);
  Node* tail = sentinel;

  for(int i = 0 ; i < k ; i++) {
      double max_score = -1.0;
      int index;
      for(int j = 0 ; j < (int)locn.size() ; j++) {
          if(scores[j] > max_score) {
              max_score = scores[j];
              index = j;
          }
      }
      Node* node = new Node(locn[index][0] , locn[index][1] , locn[index][2] , -1 , -1);
      tail -> right = node;
      node -> left = tail;
      tail = node;
      scores[index] = -1.0;
  }

  Node* head = sentinel -> right;
  delete sentinel;
  sentinel = nullptr;

  Node* current = head;
  int i = 1;
  while(current && i < k) {
      current = current -> right;
      i++;
  }
```

Significance of TF-IDF method used:

When you apply Term Frequency-Inverse Document Frequency (TF-IDF) on a word w and a paragraph p, the resulting TF-IDF score provides a measure of how important or significant that word is in the context of the specific paragraph within the entire corpus. Here's what the TF-IDF score signifies:

Term Frequency (tf(w , p)):

- Measures how often the word w occurs within the specific paragraph p.
- If w appears frequently in p, the TF value will be higher.

Inverse Document Frequency (idf(w)):

- Reflects how unique or important the word **w** is across all the paragraphs in the corpus.
- If **w** is common across many paragraphs, the **IDF** value will be lower.

TF-IDF Score (tf–idf(w , p)):

- The product of **TF** and **IDF.**
- **High TF-IDF** scores indicate that the word is both frequent in the specific paragraph and relatively unique across the entire corpus.
- **Low TF-IDF** scores suggest that the word is either common in many paragraphs or not very prevalent in the specific paragraph.

In practical terms, when you use TF-IDF for paragraph scoring, you can interpret the TF-IDF score as a weighted measure of the importance of a word within a specific context (paragraph) while considering its overall distribution across the entire corpus. It helps identify words that carry significance within specific paragraphs and might contribute more meaningfully to the content of those paragraphs.

Conclusion:

So based on the above scoring method, we have given scores to the relevant paragraphs and then sent the top_k paragraphs to the LLM.

Prompt Engineering Techniques Used:

Instead of using some complex algorithm to get us the important words, we asked ChatGPT itself for the important words, using a query.

```
void QNA_tool::query(string question, string filename){
    string API_KEY = "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";
    string q1 = "For the given sentence : \"" + question + "\", give me only the important words in lowercase and avoid the typos. Please do not write anything
    query_llm_chat(filename , API_KEY , q1);
    string a1 = read_chatgpt_response();
```

Here the query is:

```
"For the given sentence: \"" + question + "\", give me only the important words in
lowercase and avoid the typos. Please do not write anything else in the response";
```

We feed this query to GPT using the function '**query_llm_chat**':

```
query_llm_chat(filename , API_KEY , q1);
```

The code snippet for query_llm_chat:

```
void QNA_tool::query_llm_chat(string filename, string API_KEY, string question) {
    // Write the query to query.txt
    std::ofstream outfile("query.txt");
    outfile << question;
    // You can add anything here - show all your creativity and skills of using ChatGPT
    outfile.close();

    // Call api_call.py with the provided query and store the response in response.txt
    int k = 0;
    std::string command = "python " + filename + " " + API_KEY + " " + std::to_string(k) + " query.txt";

    system(command.c_str());
}
```

This C++ function, query_llm_chat, performs the following tasks:

1.Write Query to File:

- Takes a string 'question' and writes it to a file named **"query.txt".**

2. Prepare and Execute Python Command:

- Forms a command string for executing a Python script (filename) along with an API key (API_KEY) and the file containing the query ("query.txt").
- The command is constructed as: "python " + filename + " " + API_KEY + " " + std::to_string(k) + " query.txt".

3.Execute Python Script:

- Calls the system command using system(command.c_str()) to execute the constructed Python command.

In summary, this function facilitates the interaction with a Python script (filename) that likely interacts with ChatGPT or a similar language model using an API key (API_KEY). It allows passing a question to the Python script, which, in turn, interacts with the language model and returns a response. The function encapsulates the process of writing the question to a file, preparing the command, and executing the Python script.

After this, **read_chatgpt_response()** is called:

The code snippet for read_chat_gpt_response():

```
string QNA_tool::read_chatgpt_response() {
    string filename = "response.txt";
    ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error: Unable to open the file " << filename << std::endl;
        return "";  // Return an empty string on error
    }

    stringstream buffer;
    buffer << file.rdbuf();
    file.close();

    return buffer.str();
}
```

This C++ function, read_chatgpt_response, accomplishes the following:

1.Specify Response File:

- Defines the filename ("response.txt") from which the response will be read.

2. Open and check File:

- Attempts to open the specified file ("response.txt").
- If the file cannot be opened, it prints an error message to the standard error stream (‘std::cerr’) and returns an empty string.

3.Read File Content:

- If the file is successfully opened, it creates a **stringstream** named **buffer**.
- Reads the entire content of the file into the ‘**buffer**’ using ‘**file.rdbuf()**’.

4.Close File:

- Closes the file to free up system resources.

5.Return Response String:

- Returns the content of the file (the response from ChatGPT) as a '**string**'.

In summary, this function facilitates reading the response generated by ChatGPT from the file "response.txt." It handles potential errors, such as being unable to open the file, and returns the response content as a string.

Similarly, we even ask ChatGPT to recommend the number of paragraphs we should give it for a proper response.

```
string q2_1 = "I have a very huge corpus of collection of Gandhi's books. I have a question and its answer lies within the corpus. ";
string q2_2 = "I can not feed the whole corpus to you and hence based on the question : \"" + question + "\" tell me how many paragraphs should ";
string q2_3 = "I feed to you in order to get a very relevant answer. Your answer must be in range 3 - 6 and only respond with a number nothing else.";
string q2 = q2_1 + q2_2 + q2_3;
query_llm_chat(filename , API_KEY , q2);
string a2 = read_chatgpt_response();


int k = stoi(a2);
Node* root = get_top_k_para_ultra(a1 , k);
query_llm(filename , root , k , API_KEY , question);
string a3 = read_chatgpt_response();
deleteList(root);
```

```
"I have a very huge corpus of collection of Gandhi's books. I have a question and its
answer lies within the corpus."
"I cannot feed the whole corpus to you and hence based on the question: \"" + question +
"\" tell me how many paragraphs should."
"I feed to you in order to get a very relevant answer. Your answer must be in range 3 - 6
and only respond with a number nothing else."
```

Above it the collective query we ask GPT, to recommend the no. of paragraphs to be given.

Below is the **api_call.py file:**

```
if __name__ == '__main__':

    # python3 <filename> API_KEY num_paragraphs query.txt
    if len(sys.argv) < 4:
        print("Usage: python api_call.py API_KEY num_paragraphs query.txt")
        sys.exit(1)

    # Read the API key from the command line
    openai.api_key = sys.argv[1]
    num_paragraphs = int(sys.argv[2])
    # print(num_paragraphs)

    # Specify the directory where files are located
    base_directory = 'C:\IIT DELHI\SEM_3\COL106\Assignments\gradescope runner A7'

    # Read the paragraphs from the files
```

```
    paragraphs = []

    for i in range(num_paragraphs):
        filename = os.path.join(base_directory, 'paragraph_' + str(i) + '.txt')
        print(filename)
        with open(filename, 'r') as f:
            paragraphs.append(f.read())
            paragraphs.append('\n')

    # add query
    query_file = os.path.join(base_directory, sys.argv[3])
    with open(query_file, 'r') as f:
        query = f.read()
        paragraphs.append(query)
        paragraphs.append('\n')

    # convert paragraphs to a single string
    paragraphs = '\n'.join(paragraphs)

    print(paragraphs)

    response = openai.Completion.create(
        engine="text-davinci-003",  # Choose the appropriate engine
        prompt=paragraphs,
        temperature=0.7,
        max_tokens=150
    )

    reply = response.choices[0].text.strip()
    print(reply)

    # Write the response to a file
    with open(os.path.join(base_directory, "response.txt"), "w") as response_file:
        response_file.write(reply)
```

There were not many changes, The only change is we are writing the reply into a **response.txt**, to get the important words and the no. of paragraphs to enter.