COL106 Assignment 7 Report

Dhruv Joshi(2022EE32079)
Khushi Memon(2022EE11695)
Simran Meena(2022EE11188)
Piyush Abnave(2022MT11289)

November 15, 2023

1 Introduction

The following implementation for the corpus file: ”The Collected Works of Mahatma Gandhi” uses Okapi BM25, a modified and simpler version of the BM25 ranking function algorithm. A query is initially provided to the algorithm, along with the entire corpus. The scores of the paragraphs in the corpus based on the relevant words extracted from the query are calculated and the paragraphs are ranked according to their importance and relevance. Based on the average length of the paragraphs and the word limit of the API, a certain number of paragraphs are chosen. These paragraphs are fed to an LLM along with the initial query. The LLM returns the appropriate replies to those queries.

The formula used is logarithmic and comprises two components. The idf (inverse document frequency) for a single word is given as

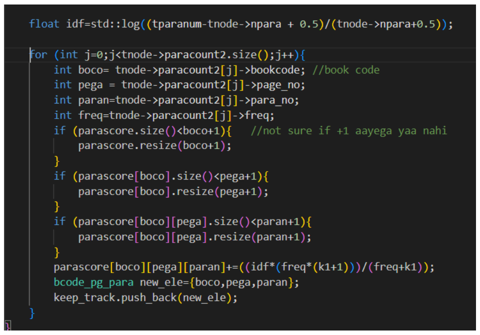
$$idf = log(\frac{N - df + 0.5}{df + 0.5}) \tag{1}$$

where N denotes the total number of paragraphs and df(document frequency) denotes the number of paragraphs containing the certain word. Using equation (1), the BM25 (score for a paragraph) is calculated by

$$BM25 = \sum idf * \frac{f_i * (k + 1)}{f_i + k} \tag{2}$$

where f denotes the frequency of words in the paragraph and k is the tuning factor. The tuning factor controls how much emphasis is given to the term frequency component in the formula. The term k+1 controls the non-linear impact of term frequency. The k term in the denominator also serves as a normalizing factor to prevent the term frequency component from dominating the score, especially for documents with high term frequencies.

The idf signifies the informativeness of a term. The higher the frequency, the lower its importance. Due to the possibility of the number of words in the paragraph being very large, we take the logarithm to dampen that effect. The following is the implementation of the above formula in the code.



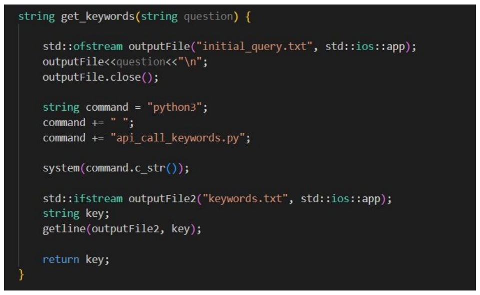
```
float idf=std::log((tparanum-tnode->npara + 0.5)/(tnode->npara+0.5));

for (int j=0;j<tnode->paracount2.size();j++){
    int boco= tnode->paracount2[j]->bookcode; //book code
    int pega = tnode->paracount2[j]->page_no;
    int paran=tnode->paracount2[j]->para_no;
    int freq=tnode->paracount2[j]->freq;
    if (parascore.size()<boco+1){   //not sure if +1 aayega yaa nahi
        parascore.resize(boco+1);
    }
    if (parascore[boco].size()<pega+1){
        parascore[boco].resize(pega+1);
    }
    if (parascore[boco][pega].size()<paran+1){
        parascore[boco][pega].resize(paran+1);
    }
    parascore[boco][pega][paran]+=((idf*(freq*(k1+1)))/(freq+k1));
    bcode_pg_para new_ele={boco,pega,paran};
    keep_track.push_back(new_ele);
}
```

Figure 1: BM25 formula to calculate the scores of the paragraphs.

2 Optimizations and Modifications

In part one of the assignment, the scores were dependent solely on the frequency of the words present in the paragraphs. However, this method of calculating scores for the paragraphs isn't very efficient. Some queries gave non-relevant paragraphs higher scores than others. So, the main focus of the optimization was to churn out more relevant paragraphs by assigning them higher scores.

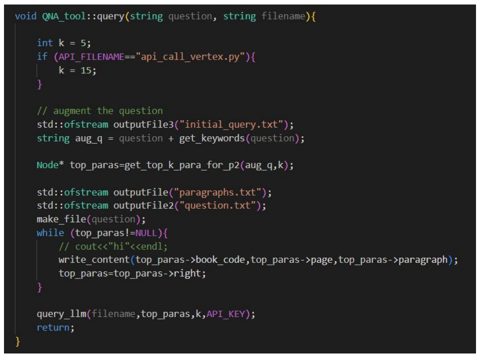
The formula(given in Part 1) was optimized to arrive at equation (2) mentioned above. The queries along with the top paragraphs obtained were passed on to PaLM API. After passing the queries and paragraphs to the API, three keywords relevant to the query were generated.



```cpp
string get_keywords(string question) {

    std::ofstream outputFile("initial_query.txt", std::ios::app);
    outputFile<<question<<"\n";
    outputFile.close();

    string command = "python3";
    command += " ";
    command += "api_call_keywords.py";

    system(command.c_str());

    std::ifstream outputFile2("keywords.txt", std::ios::app);
    string key;
    getline(outputFile2, key);

    return key;
}
```

Figure 2: function generates keywords

The generated keywords are now added to the final query list and the list is used to find the top k relevant paragraphs. The query list and top k paragraphs were passed in Chat GPT as queries and context respectively. We set the value of k to 5 for this corpus. The context was given as a background, and strictly based on the context given, a relevant reply to the passed query was generated.



```cpp
void QNA_tool::query(string question, string filename){

    int k = 5;
    if (API_FILENAME=="api_call_vertex.py"){
        k = 15;
    }

    // augment the question
    std::ofstream outputFile3("initial_query.txt");
    string aug_q = question + get_keywords(question);

    Node* top_paras=get_top_k_para_for_p2(aug_q,k);

    std::ofstream outputFile("paragraphs.txt");
    std::ofstream outputFile2("question.txt");
    make_file(question);
    while (top_paras!=NULL){
        // cout<<"hi"<<endl;
        write_content(top_paras->book_code,top_paras->page,top_paras->paragraph);
        top_paras=top_paras->right;
    }

    query_llm(filename,top_paras,k,API_KEY);
    return;
}
```

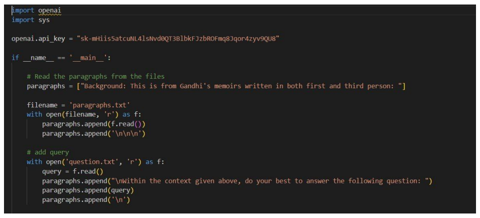
Figure 3: paragraphs and query fed to the LLM returns the answer in an output file "filename".

The reply generated for the query proved to be accurate, as in case the context doesn't have a justified reply to the query, our model mentions that the context provided is insufficient to provide a reply to the query.

3 Changes made to API Calls

We have 3 different options for API calls:

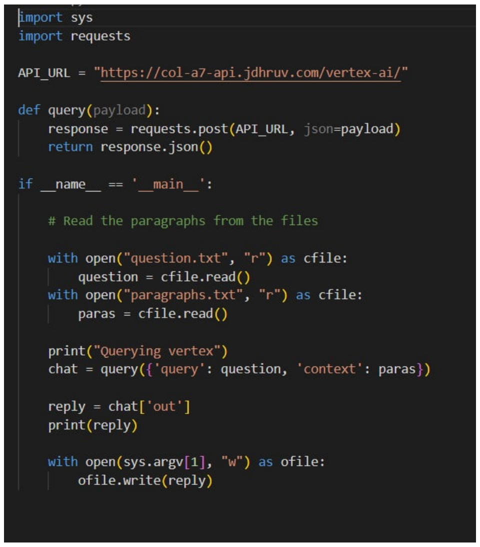
1. `api_call.py`: This is the regular GPT-3.5 API call



```python
import openai
import sys

openai.api_key = "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"

if __name__ == '__main__':

    # Read the paragraphs from the files
    paragraphs = ["Background: This is from Gandhi's memoirs written in both first and third person: "]

    filename = 'paragraphs.txt'
    with open(filename, 'r') as f:
        paragraphs.append(f.read())
        paragraphs.append('\n\n\n')

    # add query
    with open('question.txt', 'r') as f:
        query = f.read()
        paragraphs.append("\nWithin the context given above, do your best to answer the following question: ")
        paragraphs.append(query)
        paragraphs.append('\n')
```

Figure 4: calls the GPT-3.5 API.

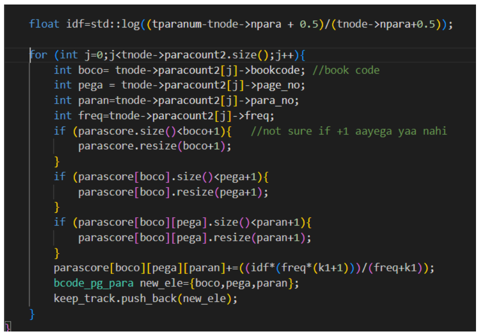
2. `api_call_vertex.py`: This calls Google's PaLM LLM API, through our own custom API endpoint at http://col-a7-api.jdhruv.com/docs



```python
import sys
import requests

API_URL = "https://col-a7-api.jdhruv.com/vertex-ai/"

def query(payload):
    response = requests.post(API_URL, json=payload)
    return response.json()

if __name__ == '__main__':

    # Read the paragraphs from the files

    with open("question.txt", "r") as cfile:
        question = cfile.read()
    with open("paragraphs.txt", "r") as cfile:
        paras = cfile.read()

    print("Querying vertex")
    chat = query({'query': question, 'context': paras})

    reply = chat['out']
    print(reply)

    with open(sys.argv[1], "w") as ofile:
        ofile.write(reply)
```

Figure 5: FastChat open source chat-box.

3. api_call_fastchat.pu: This calls a HuggingFace model through our own custom api at http://col-a7-fastchat.jdhruv.com/docs



```cpp
    float idf=std::log((tparanum-tnode->npara + 0.5)/(tnode->npara+0.5));

    for (int j=0;j<tnode->paracount2.size();j++){
        int boco= tnode->paracount2[j]->bookcode; //book code
        int pega = tnode->paracount2[j]->page_no;
        int paran=tnode->paracount2[j]->para_no;
        int freq=tnode->paracount2[j]->freq;
        if (parascore.size()<boco+1){   //not sure if +1 aayega yaa nahi
            parascore.resize(boco+1);
        }
        if (parascore[boco].size()<pega+1){
            parascore[boco].resize(pega+1);
        }
        if (parascore[boco][pega].size()<paran+1){
            parascore[boco][pega].resize(paran+1);
        }
        parascore[boco][pega][paran]+=((idf*(freq*(k1+1)))/(freq+k1));
        bcode_pg_para new_ele={boco,pega,paran};
        keep_track.push_back(new_ele);
    }
}
```

Figure 6: FastChat open source chat-box.

Additionally, for prompt engineering, each search term is run through PaLM LLM in order to extract 3 keywords (may or may not be in the search term). These are appended to the end, and common words are removed from the search before scoring each paragraph with the search term, enhancing the search massively.

The custom API endpoints are running on a server on Google Cloud (with 1x NVIDI L4). This is free for new users as GCloud offers 300 dollars of credit to new accounts

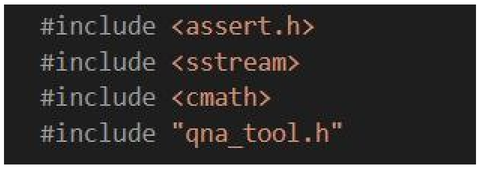
4 The HuggingFace model

The open-source HuggingFace model used is Vicuna-7B by FastChat, which is available both on their GitHub as well as on HuggingFace. It is essentially a fine-tuning of Facebook's LLaMA model with 7 billion parameters on public data. This allows it to give ChatGPT-like quality results.

This model is running locally on the Google Cloud machine using GPU and has an API endpoint compatible with the OpenAI library. Cloudflare Tunnels is used to map this API endpoint to the URL http://col-a7-fastchat.jdhruv.com/docs and `api_call.py` uses the OpenAI library to call it and prints the result.

5 Additional Headers

cmath.h was included to access the logarithmic function.



```
#include <assert.h>
#include <sstream>
#include <cmath>
#include "qna_tool.h"
```

Figure 7: cmath header file

6 Instructions to run the model

To run the model, click here and follow the given instructions.