Search Enigine - Help Doc

Hey everyone,

I hope you all are doing great. I have received several doubts regarding the exact process one should follow while making the search engine. So in this doc, I am sharing an overview of one of the approaches to making it. (This will be in MERN but the concepts could be implemented in any tech stack.) You can very well tweak this approach or use any other approach, just the end result should be a working search engine.

Pre-requisites for understanding the below approach:
Web-Scrapping, TF-IDF Algorithm [The topics have been discussed in detail in the sessions]

Part-1:

Since our Search Engine needs a database and there is no readily available database for the same, we need to build it on our own. So we will scrape the questions from the internet.
For scrapping, you can use any website of your choice.
Scrapping Leetcode is a bit straightforward I feel. Also, I have shown how to scrape CodeChef in the session and provided the code.
Keep the scrape count to be around 2.5k-3k.
While scrapping makes sure you are maintaining an organized file structure and follow a naming convention.

One of the naming conventions which I have followed is:

- problem_urls.txt
- problem_titles.txt
- problems
    - problem_text_1.txt
    - problem_text_2.txt
    - problem_text_3.txt
    - problem_text_4.txt
    - ….. And so on

Here,
problem_urls.txt -> Text file containing the URLs of all the scraped problems separated by ‘\n’
I.e.

```
https://leetcode.com/problems/two-sum
https://leetcode.com/problems/regular-expression-matching
https://leetcode.com/problems/same-tree
https://leetcode.com/problems/minimum-cost-to-merge-stones
https://leetcode.com/problems/grid-illumination
…
```

problem_titles.txt -> Text file containing the Titles of all the scraped problems separated by ‘\n’
I.e.

```
Two Sum
Regular Expression Matching
Same Tree
Minimum Cost to Merge Stones
Grid Illumination
…
```

The line number also indicated the problem number in our naming convention for each problem
I.e.
problem_text_1.txt contains the content of the first problem which is “Two Sum” in the above example and so on.

Note:

- While scrapping Leetcode you might face an issue that some problems are premium and might not open.
  Hint to identify that: The premium problems HTML doesn’t have the div with class “description__24sA” which contains the problem description
- Suppose you have scrapped from multiple websites, then what you can do is scrape them separately and then append their URLs and Titles below the problem_urls.txt and problem_titles.txt. Also, start the problem counter (used while naming the files) on the second website after the number of problems scrapped from the first website.

  E.g. Suppose you scrapped the leetcode first and you extracted say 1563 problems from it. Now when you move on to scrapping let's say CodeChef, start its problem counter after 1563 so that the scrapped questions are stored in file names - problem_text_1564.txt, problem_text_1565.txt etc
- Your scrapped problems can have some words in unicode like \x8\x6.. Or some special symbols represented by ?. But their is no need to worry about them as user wont be able to write those keywords during query anyways so they will be never be hit and wont contribute.

Part 2:

Once we have all our questions scrapped, now we need to work on generating the TF-IDF matrix of our documents. Our aim is to generate the following files:

- keywords.txt -> Contains all the unique keywords (across all the documents) in a sorted way.
  I.e.
  add
  aim
  ball
  bipartite
  … ans so on

  (This is just an example)

- IDF.txt -> Contains the IDF value of each keyword (in the same order as in keywords.txt)
  I.e.
  3.456
  4.2234
  13.42
  2.345
  … and so on

  (Note: Use the formula 1+log(N/nt) [Base 10] for IDF calculation)

- TFIDF.txt-> Contains the TFIDF matrix values. Now since each document will have a vector size in 10s of thousand (As total number of distinct keywords can be quite high), so instead of storing all the TFIDF values we just store the non zero ones with their i and j value (so that when we read this file we can place the value of this TFIDF on the respective i and j)
  I.e
  1 456 4.24
  1 579 2.456
  2 1099 4.56
  .. and so one

  This means:
  TFIDF[1][456] = 4.24
  TFIDF[1][579] = 2.456
  Now since all 1s are done, by default all other values of doc 1 will be zero
  TFIDF[2][1099] = 4.56

- Magnitude.txt -> Contains the magnitude of TDIDF vector of each document. Since while calculating similarity we will need to divide by the magnitude of each vector hence we can precompute it and store it: [Each line represents the value of magnitude of ith document]
  12.45
  14.2
  145.24
  45.13
  … and so on

Now before you move on to making the txt files, make sure that you check the correctness of your algorithm by calculating the result for a sample query.

While doing this convert the query keywords to lowercase too as tWo SuM must be treated as “two” and “sum”

Note:

- While making the keywords array and the TF-IDF array convert all the keywords of each document to lowercase while processing as this will ensure that Sum and sum are considered as same.

Part 3:

We will now proceed to make our web application using Node.js and Templating.

- You need to create a express server with a get route to “/”.

- Now on this route you need to create a form in which user can enter the query and once user hits submit your website should fire a request to “/search” with the query string attached as a query parameter. So the actual request will be:

  GET /search?query=get the sum of all the elements of an array [Just an example]

- Now your express app should handle this route and return the top 5 similar documents to the frontend as response to display.

  For handling before making the request (or just with the start of your app) you should read all the txt files and generate the corresponding arrays. E.g. TFIDF Matrix, Keywords Array, IDF Array, Magnitudes Array, N (Total Number of documents), W (Total number of unique keywords).
  Now use these arrays for calculating similarity values in the route handler for /search route and return the top 5-10 results as response to the user.

  You can watch the following playlist (VIdeo 24-37) for how to work with query strings, frontend etc.:
  https://youtube.com/playlist?list=PL4cUxeGkcC9gcy9lrvMJ75z9maRw4byYp

  They are a bit old videos, but the concept remains the same. So first watch the Node playlist which I sent earlier to learn the concepts of Node.js and Express.js

Note:

If your search is taking a lot of time to execute maybe try to reduce the number of documents to say 2000 and then generate the files again.

Read this document by 20th evening and try to build it. On 20th we will have our final session for some doubts and how to deploy your node application. After that you will be given 2 Days for completing and wrapping up your search engine. Details regarding the submission format will be communicated after the session.