Research First Task

jialen2

October 2021

The problem I will solve in this task is that given a set of concept terms, a set of documents and a query, find the top 3 concept terms for the query in the documents, by evaluating how frequently the query and the concept terms appear together and how close they are. I will solve this problem in two different conditions.

First, the user query is inside the concept terms set, or it is relevant to some terms in the concept terms set.

My work in this condition can be divided into two part.
1. Pre-compute the co-occurrence matrix.
The program first loads all concept words and all document (both total number of 10000 in my case). Then for each query, it searches in the document sets to find out which documents contain the query. We use a 2-D array *df_table* to store all information, with each entry (i, j) represents whether query i appears in document j. Then we calculate $df\_table \times df\_table.\mathrm{T}$ to get the co-occurrence table with each entry (i, j) represents how many documents contain both concept i and concept j. For each row in the co-occurrence table, we find the top 5 most relevant concepts with the concept in the current row. Then we searches both concepts in the documents and find out documents containing both of them. We evaluate the distance between two words by calculating their index difference in the documents. Then we adjust the score by

$$new\_score = original\_score + \log_2 \frac{1000}{\frac{total\_distance}{num\_document+1} + 1} \tag{1}$$

where the *total_distance* is the distance sum in all relevant documents and *num_document* is the total number of relevant document. We then get the top 3 concept words based on the adjusted score and store them inside the file "*top_3_concepts*" for later use. Also, we cut at most 3 subsequence that contains both concept words in the document and store them inside the file "*store_snippet*"

The time complexity of this process can be analyzed as follows:
Suppose the average document length is l, the total number of document is d, the total number of concept words is c. The dominant term for runtime in this

algorithm is c*l*d + $c^{2.37}$. c*l*d is the runtime for searching c concepts words in all document(I'm not sure how the indexing and searching algorithm works in the whoosh library, so I will just use the worst case runtime O(n) as the time complexity). $c^{2.37}$ is the runtime for matrix multiplication(by the latest runtime) for getting the co-occurrence table, would be O(c*l*d + $c^{2.37}$).
The second part is to use the results from the first part to get the snippets and the most related concepts. If the query is in the concept words set, we will load data in the "*top_3_concepts*" file and the top 3 most related concept words and load data in "*store_snippet*" to get snippets. If the query is not in the concept words set, the program will try to find a substring or superstring of the query and use that instead. If we can't find that, the program will raise an error. The time complexity of this process is O(c), since "*store_snippet*" and "*top_3_concepts*" each have 3c and c rows, so loading from the files takes O(c) time.
In general, we will use O(c*l*d + $c^{2.37}$) time for pre-computing and O(c) for searching. The pre-computing process just needs to be done for once, then user can use O(c) to search each time.

The second condition is that the user query is highly irrelevant to the concept terms set(e.g. most concept terms are from the field of technology but the user query is about food or entertainment)

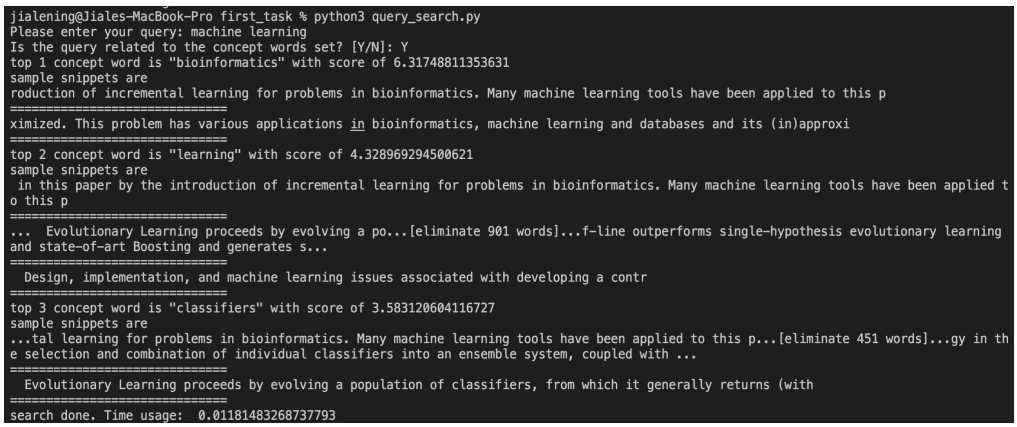
In this case, The program in *input_parser.py* file will first first split all documents into sentences, then search in the sentences set with concept terms, and only output the sentences with concept terms in it. The program will store all these sentences inside the file "*concept_words_sentences*". Then when user tries to search query, the program will only search through the sentences containing the concept terms instead of the whole document set. The program will compute for each concept term, how many times it appears in the same sentence as the query(which I will refer to as CF(concept frequency) later), then compute the final score for a concept word to a query using the formula Score = CF*$\log_2 \frac{ND+1}{DF+1}$, where DF is the number of documents that contain the concept word, and ND is the total number of document. This formula will penalize the word appears too frequently in all documents, so that only the most relevant concept terms will remain. Then the program will find out the top 3 highest score concept terms and print out the sentence containing both terms as the snippet of the query and the concept term.

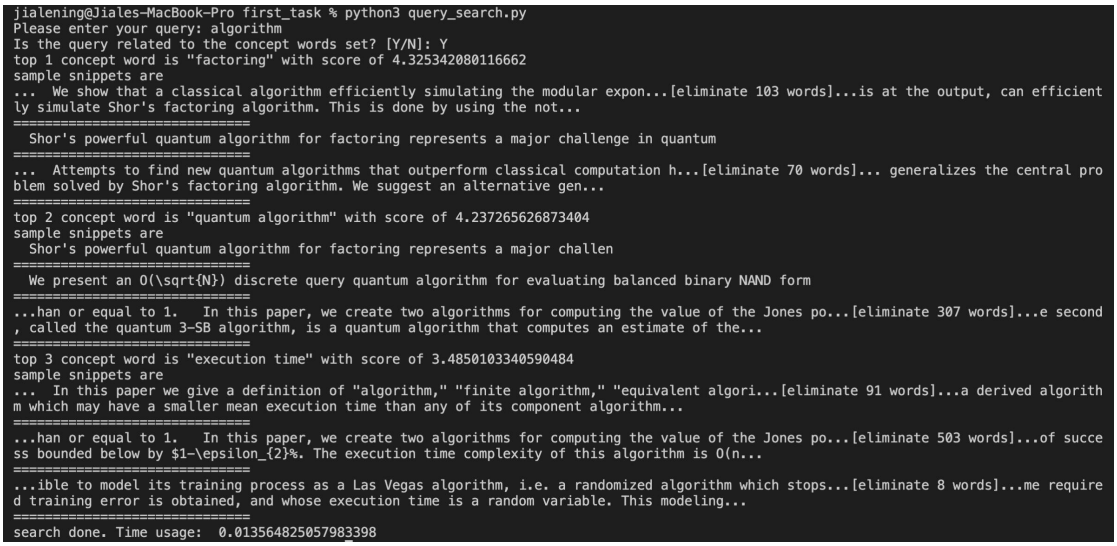
The time complexity for this algorithm is relatively larger than the previous one. The pre computation part take O(c*l*d) since we need to search through the whole document set to find sentences containing each concept terms. When a user searches a query, we need to search through the whole document sets in the worst case, which takes O(l*d) time.

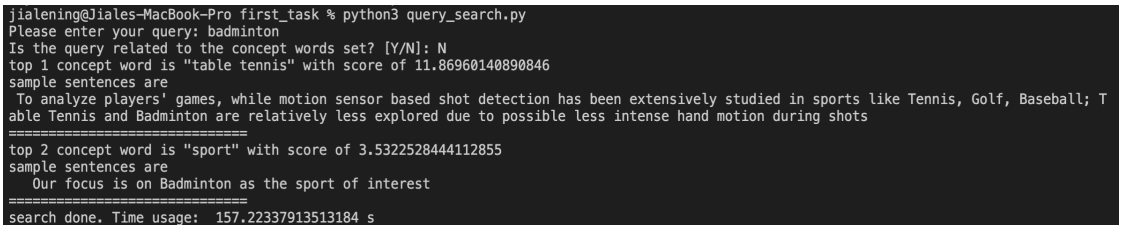
Analysis: The first algorithm is much faster than the second algorithm when we search queries. The first takes less than 1 second to get the result while the second takes about 2.5 min to take results. But we can only use the first algo-

rithm if our query is inside or similar to the concept words set. Both approaches return results as I expect.

Some sample results are listed below:
The first two images are searching concept-related query. I chose "machine learning" and "algorithm" as examples. The last one is searching unrelated query. I chose "badminton" as an example.







Paper Analysis

The problem described in the paper is given a query image, examine the most similar texture image from the image database.

Key ideas:

1. Compute the Grey Level co-occurrence matrix(GLCM) for each image. Each entry (i, j) in the matrix represents the number of occurrence for grey level i and j that are a distance d apart in the origin image.
2. Four properties of an image can be calculated from each GLCM, and the similarity between images can be calculated by the Euclidean distance of the property vectors of two images.
3. Evaluation of this experiment shows that the results could be less reasonable if the image contains large area of other textures. The method to solve this problem is to split the image into 9 blocks and apply retrieve procedure to them. The result is better.

Link for the paper:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.1056rep=rep1type=pdf