CSE 435/535 Information Retrieval (Fall 2024)
Project 2: Boolean Query and Inverted Index

Due Date: Oct 30, 2024, before 11:59 PM

Overview

In this project, you will be given a sample input text file consisting of Doc IDs and sentences. Based on this provided input text file, your task is to build your own inverted index and host it as a Flask app for querying. Your index should be stored as a Linked List in memory as the examples shown in the textbook (refer to Chapter 1 – Boolean Retrieval). Having built this index, you are required to implement a Document-at-a-time (DAAT) strategy to return Boolean query results. Your implementation should be based only on Python3.

Input Dataset

input_corpus.txt is a tab-delimited file where each line is a document; the first field is the document ID, and the second is a sentence. The corpus can be found in the data directory (contributed by former UB student and instructor Sougata Saha) at this link https://github.com/proto-ai-lab/CSE_4535_Fall_2021/tree/master/project2. We will be using this corpus for project 2 all throughout (even for grading).

Note: we will use only the data folder of the Github repo linked above. You can use the template code as a reference, but you need to develop your own codebase to complete this project.

Example:

```
113257    COVID-19 and diabetes mellitus: implications for prognosis and clinical management
156757    Seroprevalence of Rodent Pathogens in Wild Rats from the Island of St. Kitts, West Indies
50439     Prevalence and genetic diversity analysis of human coronaviruses among cross-border children
```

Project Requirements

Part 1: Build Your Own Inverted Index

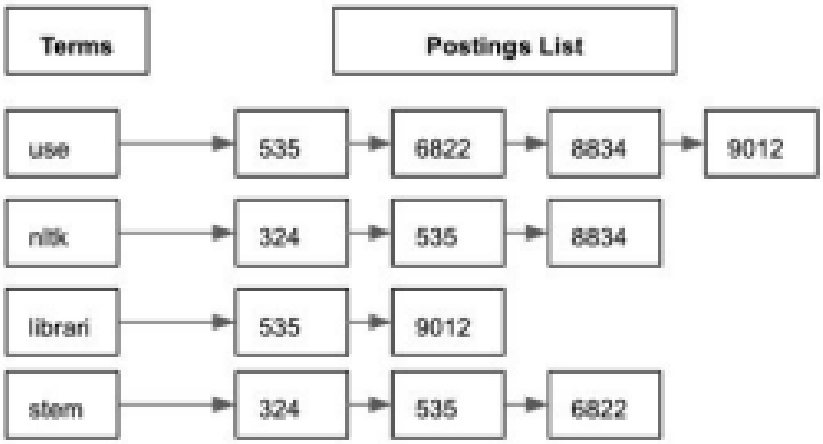
Implement a pipeline that takes as input the given corpus and returns an inverted index.

1. Extract the document ID and document from each line.
2. Perform a series of preprocessing on the document:
    a. Convert document text to lowercase.
    b. Remove special characters. You can replace them by the space char. Only alphabets, numbers, and whitespaces should be present in the document.
    c. Remove excess whitespaces. There should be only 1 white space between tokens, and no whitespace at the start or end of the document.
    d. Tokenize the document into terms using a white space tokenizer.
    e. Remove stop words from the document.
    f. Perform Porter's stemming on the tokens.
    g. Sample preprocessing output.

```
Input ->  535   You can  use NLTK(library)   for stemming   !

Doc id ->  535
Doc text ->  You can  use NLTK(library)   for stemming   !

Doc text post processing ->  you can use nltk library for stemming

Doc tokens post whitespace tokenizing ->  ['you', 'can', 'use', 'nltk', 'library', 'for', 'stemming']
Doc tokens post stopword removal ->  ['use', 'nltk', 'library', 'stemming']
Doc tokens post stemming ->  ['use', 'nltk', 'librari', 'stem']
```

Note: **Do not use NLTK** or any other package for **whitespace tokenization** or **special character removal**. You might end up getting different results. Usage of simple python **regex** and inbuilt functions are recommended. You can use NLTK for stemming.

3. For each token, create a postings list. Postings list must be stored as **linked lists**. Postings of each term should be ordered by increasing document ids.



4. Add skip pointers in the postings lists for each term. There should be floor(sqrt(L)) skip connections (floor(sqrt(L))-1 connections in case L is a perfect square) in a postings list, and the length between two skips should be round(sqrt(L), 0), where L = length of the postings list.

5. Add tf-idf scores to each element in the postings list. The tf-idf should be calculated using the following formulas:

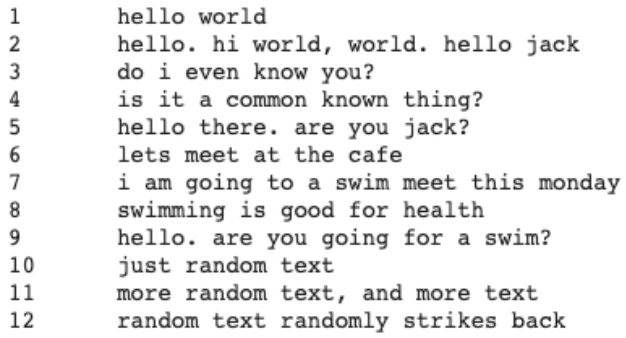
Tf = (freq. of token in a doc after pre-processing / total tokens in the doc after pre-processing)

Idf = (total num docs / length of postings list)

tf-idf = Tf*Idf

Part 2: Boolean Query Processing

You are required to implement the following methods and provide the results for a set of Boolean “AND” queries **on your own index**. Results should be output as a JSON file in the required format. For the rest of the documentation, we are going to assume the below corpus & queries.



```
1       hello world
2       hello. hi world, world. hello jack
3       do i even know you?
4       is it a common known thing?
5       hello there. are you jack?
6       lets meet at the cafe
7       i am going to a swim meet this monday
8       swimming is good for health
9       hello. are you going for a swim?
10      just random text
11      more random text, and more text
12      random text randomly strikes back
```

```
hello world
hello swimming
swimming going
random swimming
```

Corpus **Queries**

1. **<u>Query Processing</u>**

   Given a user query, the first step must be **preprocessing the query** using the same document preprocessing steps.

   a. Convert query to lowercase.

   b. Remove special characters from query. You can replace them by the space char

   c. Remove excess whitespaces. There should be only 1 white space between query tokens, and no whitespace at the starting or ending of the query.

   d. Tokenize the query into terms using a white space tokenizer.

   e. Remove stop words from the query.

   f. Perform Porter's stemming on the query tokens.

   g. Sample query processing:

```
Input query ->  Which library to use for stemming?
Query post processing ->  which library to use for stemming

Doc tokens post whitespace tokenizing ->  ['which', 'library', 'to', 'use', 'for', 'stemming']
Doc tokens post stopword removal ->  ['library', 'use', 'stemming']
Doc tokens post stemming ->  ['librari', 'use', 'stem']
```

2. **Get postings lists**

This method retrieves the postings lists for each of the given query terms. The input of this method will be a set of terms: term0, term1,..., termN. It should output the **document ID wise sorted** postings list for each term. Below is a sample format of the same:

```
'postingsList': {
    'go': [7, 9],
    'hello': [1, 2, 5, 9],
    'random': [10, 11, 12],
    'swim': [7, 8, 9],
    'world': [1, 2]
},
```

3. **Document-at-a-time AND query without skip pointers**

This function is used to implement multi-term Boolean AND query on the index using document-at-a-time(DAAT) strategy. The input of this function will be a set of query terms: term0, term1, ..., termN. You will need to implement the MERGE algorithm and **return a sorted list of document ids, along with the number of comparisons made**. Below is a sample of the output:

```
'daatAnd': {
    'hello swimming': {
        'num_comparisons': 6,
        'num_docs': 1,
        'results': [9]
    },
    'hello world': {
        'num_comparisons': 2,
        'num_docs': 2,
        'results': [1, 2]
    },
    'random swimming': {
        'num_comparisons': 3,
        'num_docs': 0,
        'results': []
    },
    'swimming going': {
        'num_comparisons': 3,
        'num_docs': 2,
        'results': [7, 9]
    }
},
```

Note: A comparison (for field "num_comparisons") is counted whenever you compare two Document IDs during the union or intersection operation. **Hint:** Determine the correct merge order to optimize "num_comparisons".

4. **Get postings lists with skip pointers**

This method retrieves the reachable documents using skip pointers for each of the given query terms. The input of this method will be a set of terms: term0, term1,..., termN. It should output the **document ID wise sorted** postings list, which can be accessed by using the skip pointers. Return empty list in case of no skip pointer for the postings list. Below is a sample format of the same:

```
'postingsListSkip': {
    'go': [],
    'hello': [1, 5],
    'random': [10, 12],
    'swim': [7, 9],
    'world': []
}
```

5. **Document-at-a-time AND query with skip pointers**

Leveraging the skip pointers, you will have to implement the MERGE algorithm, and **return a sorted list of document ids, along with the number of comparisons made.**

```
'daatAndSkip': {
    'hello swimming': {
        'num_comparisons': 4,
        'num_docs': 1,
        'results': [9]
    },
    'hello world': {
        'num_comparisons': 2,
        'num_docs': 2,
        'results': [1, 2]
    },
    'random swimming': {
        'num_comparisons': 2,
        'num_docs': 0,
        'results': []
    },
    'swimming going': {
        'num_comparisons': 3,
        'num_docs': 2,
        'results': [7, 9]
    }
},
```

Part 3: TF-IDF Scoring

1. **Document-at-a-time AND query without skip pointers, sorted by tf-idf**

Sort the output of DAAT without skip pointers (Part 2 step 3) using tf-idf scoring, where the tf-idf should be calculated using the formula mentioned in Part 1 step 5. The retrieved documents & number of comparisons remain the same as DAAT without skip pointers. Below is an example of the same.

```
'daatAndTfIdf': {
    'hello swimming': {
        'num_comparisons': 6,
        'num_docs': 1,
        'results': [9]
    },
    'hello world': {
        'num_comparisons': 2,
        'num_docs': 2,
        'results': [1, 2]
    },
    'random swimming': {
        'num_comparisons': 3,
        'num_docs': 0,
        'results': []
    },
    'swimming going': {
        'num_comparisons': 3,
        'num_docs': 2,
        'results': [9, 7]
    }
},
```

2. **Document-at-a-time AND query with skip pointers, sorted by tf-idf**

Sort the output of DAAT with skip pointers (Part 2 step 4) using tf-idf scoring, where the tf-idf should be calculated using the formula mentioned in Part 1 step 5. The retrieved documents & number of comparisons remain the same as DAAT with skip pointers. Below is an example of the same.

```
'daatAndSkipTfIdf': {
    'hello swimming': {
        'num_comparisons': 4,
        'num_docs': 1,
        'results': [9]
    },
    'hello world': {
        'num_comparisons': 2,
        'num_docs': 2,
        'results': [1, 2]
    },
    'random swimming': {
        'num_comparisons': 2,
        'num_docs': 0,
        'results': []
    },
    'swimming going': {
        'num_comparisons': 3,
        'num_docs': 2,
        'results': [9, 7]
    }
},
```

3. **Output Format**

The results of the postings list and DAAT AND queries must be combined in a single python dictionary, and made available through an API. You can find an example output in file ***sample_output.json***.

Part 4: Hosting the Project

You are required to expose your index using a Flask endpoint. The endpoint should be accessible via port 9999 and must be named "execute_query" (example: http://ip:port/execute_query). The endpoint must accept the following parameters as part of the payload:

1. **Queries**

A list of Boolean queries which need to be run against your index. Below is an example of a payload.

```
{
    "queries": ["hello word", "hello swimming", "swimming going", "random swimming"]
}
```

Part 5: Project Grading

1. **What to submit**

For the final submission, you are required to submit a **JSON file** containing the details of your endpoint. The JSON file **MUST** be named ***project2_index_details.json* ,** and must contain the below contents.

```
{
    "ip": "<IP address of your instance (without the angle brackets, with the quotes)>",
    "port": "<Port number through which the App is accessible (without the angle brackets, with the quotes)>",
    "name": "<Name of your endpoint (without the angle brackets, with the quotes)>"
}
```

2. **How to submit**

Submit a zip file containing the following on UB Learns.

1. The JSON file (described above)
2. The codebase in a directory named 'src' which contains all code files used for this project.

3. **Grading Rubric**

Total points for this project: **15**
We will run 3 queries against your index. Below is the grading breakdown for the same.

1. Correct retrieval of postings list: **1.5**
2. Correct retrieval of postings list with skip pointers: **3**
3. DaatAnd query: ((0.5 + 0.5)*3 = **3**)
    a. Correct documents retrieved (0.5)
    b. num_comparisons within an acceptable range of +/- 5% of desired comparisons (0.5)
4. DaatAnd query with skip pointers: ((0.5 + 1.0)*3 = **4.5**)
    a. Correct documents retrieved (0.5)
    b. num_comparisons within an acceptable range of +/- 5% of desired comparisons (1.0)

5. DaatAnd query with Tf-Idf: ((0.25 + 0.25)*3 = **1.5**)
    a. Correct documents retrieved (0.25)
    b. num_comparisons within an acceptable range of +/- 5% of desired comparisons (0.25)
6. DaatAnd query with skip pointers and Tf-Idf: ((0.25 + 0.25)*3 = **1.5**)
    a. Correct documents retrieved (0.25)
    b. num_comparisons within an acceptable range of +/- 5% of desired comparisons (0.25)

Part 6: Few Pointers and Assumptions

The following assumptions can be made:

1. The number of input queries will be 3, but the query terms can be varied.
2. All of the input query terms are selected from the vocabulary.
3. Query terms should be processed in the order in which they are written in the query. Say, you should process term0 first, then term1, so on and so forth.
4. **DO NOT** use python built-in methods to do unions and intersections on postings lists directly. Create your own Pointers/References!
5. We will use **MOSS** to check **plagiarism**.
6. **Output should be formatted exactly the same as required. Otherwise, you will not be able to get credits because grading will be automated!**

Note: Late submissions will **NOT** be accepted. Please start early.

FAQ:

1. **How should I get started on this project?**
   First, make yourself familiar with fundamental concepts such as dictionary, postings, Inverted Index, and Boolean operations. The best place to start is

reading thoroughly the lecture slides and Chapter 1,2 of the referred textbook. Also, please get familiar with Flask API. Here are some general tutorials to get you started:

Blog:

https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3

Video: https://www.youtube.com/watch?v=MwZwr5Tvyxo
Official website: https://flask.palletsprojects.com/en/2.2.x/
Reference code (optional, requirements changed): https://github.com/proto-ai-lab/CSE_4535_Fall_2021/tree/master/project2

What programming concepts are needed to complete this project?

You are expected to know how to work with functions and be familiar with basic data structures such as dictionary/hash maps, lists/arrays, Linked lists, Queue, and so on. Also, check out the python NLTK package. You will use the functions provided in NLTK for stemming and stopword removal.

2. **My program takes a while to execute. Would that be a problem in grading?**

   Although the focus of this project is to test the correctness, we encourage you to be mindful of the data structure you are using for implementation. Unless it takes an unreasonably long time, you are fine in terms of grading but again please carefully analyze your code.

3. **Will the same corpus be used during the final submission?**

   Yes

4. **Will the same queries be used during the final submission?**

   No, we will use a different set of queries.

5. **Will more than 3 queries be used during the final submission?**

   No, we will use exactly 3 queries to test your index.

6. **How many terms will there be in each query?**

It will vary. There is no limit on the number of terms a query can have.

7. **Should the keys in the postingsList be preprocessed or raw?**

   The keys in the postings list are the pre-processed tokens. If there are stopwords in the input query, it should not be a part of the postings list.

8. **Should the keys in the daatAnd and other boolean queries be preprocessed or raw?**

   The keys must be the raw input query.

9. **What should I return for postingsListSkip, in case a postings list is too small to have skip pointers?**

   Return an empty list []

10. **What should I return if the DAAT AND query did not find any documents?**

    Return an empty list [] as results, with num_docs as 0. The num_comparisons should not be 0.

11. **Can I use Python lists instead of linked list?**

    No. Not using a linked list will get zero points.

Acknowledgement: Thanks to former UB instructor Sougata Saha for his contribution to this project.