

CSE 435/535 Information Retrieval (Fall 2021)

Project 2: Boolean Query and Inverted Index

Due Date: October 17th 2021, 11:59 pm (EST)

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 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. We will be using this corpus for project 2 all throughout (even for grading).

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 which 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. 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 starting or ending of the document.
 - d. Tokenize the document into terms using 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 !

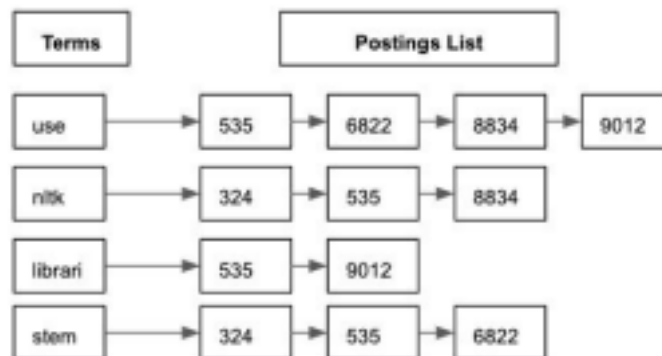
Doc text post processing -> you can use nltk library for stemming

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', 'sten']

```

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 $\text{floor}(\sqrt{L})$ skip connections ($\text{floor}(\sqrt{L})-1$ connections in case L is a perfect square) in a postings list, and the length between two skips should be $\text{round}(\sqrt{L}, 0)$, 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:

$\text{Tf} = (\text{freq of token in a doc after pre-processing} / \text{total tokens in the doc after pre-processing})$

$\text{Idf} = (\text{total num docs} / \text{length of postings list})$

$\text{tf-idf} = \text{Tf} * \text{Idf}$

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?	hello world
10	just random text	hello swimming
11	more random text, and more text	swimming going
12	random text randomly strikes back	random swimming

Corpus

Queries

1. Query Processing

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
- 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 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. 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. Input of this function will be a set of

query terms: term0, term1, ..., termN. You will have 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. 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 and 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 point 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 point 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.

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 needs to be run against your index.

2. Random Command

A random command from the grader, which sanity checks the implemented backend data structures.

Below is an example of a payload


```
{
  "queries":["hello world", "hello swimming", "swimming going", "random swimming"],
  "random_command":"self.indexer.get_index()['random'].traverse_list()"
}
```

Part 5: Project Grading

1. What to submit

An auto grader (which will be made available to you in a week) will be used to evaluate your index via the Flask app you create. For the final submission, you are required to submit a **pickle file (not a json file, not a text file)** containing the details of your endpoint. The pickle file **MUST** be named *project2_index_details.pickle*, and must contain the below contents.

```
1 {
2   "ip": "<IP address of your instance (without the angle brackets, with the quotes)>",
3   "port": "<Port number through which the App is accessible (without the angle brackets, with the quotes)>",
4   "name": "<Name of your endpoint (without the angle brackets, with the quotes)>"
5 }
```

2. How to submit

Similar to project 1, we will use Timberlake for submissions. You must ssh into timberlake.cse.buffalo.edu, and use the command *submit_cse535 project2_index_details.pickle* or *submit_cse435 project2_index_details.pickle* for grad/undergrad students respectively.

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: $(0.5 * 3 = 1.5)$
2. Correct retrieval of postings list with skip pointers: $(1 * 3 = 3)$
3. DaatAnd query: $((0.5 + 0.5) * 3 = 3)$
 - a. Correct documents retrieved (0.5)
 - b. num_comparisons within 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 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 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 acceptable range of +/- 5% of desired comparisons (0.25)

Note: You will get 0 points if the sanity_checker detects inconsistencies in the data structures used.

Part 6: Getting Started

You can use the starter codebase for the project, which is available here: https://github.com/proto-ai-lab/CSE_4535_Fall_2021.git -> project2. The codebase has several pre-implemented methods (creating a Flask app, final formatting, etc.), which you can use. If you do intend to use the codebase for this project, then please **DO NOT** modify the portions of the code marked as “**DO NOT MODIFY**”. Only implement the functions marked with “**To be implemented**”, and add more functionality to the codebase if needed. In case you choose not to use the codebase (which is not recommended), please make sure you adhere to the output formatting requirements. Failing to do so will result in exceptions in the auto-grader, and you will lose points.

Part 7: 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 build-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. The deadline is firm (i.e., October 17th 11:59 PM (EST)), if your timestamp is 12:00 AM, it is a late submission. Please start early.

FAQ:

1. How should I get started for 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. The codebase has hints to help you in case you are stuck.

2. 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 should use the functions provided in NLTK for stemming and stopword removal.

3. 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.

4. Will the same corpus be used during final submission?

Yes

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

No, we will use a different set of query.

5. Will more than 3 queries be used during 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. I used the auto-grader before, and got a good score. However, I made a very small mistake in my final submission, and got a low score. Can my project be re-graded?

No. The auto-grader is made available to you so that you can prevent such mistakes before the deadline. Please be cognizant while submitting your project, and sanity check your submission multiple times.

12. What is the purpose of sanity_checker?

The sanity checker is used to run random commands against your project, in order to ensure (i) correct data structures have been implemented (ii) detect plagiarism in code implementation. Hence, do not change the sanity_checker function.

13. What happens if the sanity checker fails?

Failing the sanity_check means either correct data structures have not been implemented, or there is an Academic Integrity violation, or both. In all the scenarios, you will receive a zero score.

14. Can I use Python lists instead of linked list?

No. Not using linked list will cause the sanity checker to fail, thus resulting in a zero score.

15. Where can I file the document corpus, and how will I validate my implementation?

1. The corpus is present in the data folder of the project2 Github repo.
Filename: *input_corpus.txt*
2. The 3 queries which the auto-grader will run during the development phase are present in the data folder of the project2 Github repo.

Filename: *queries.txt*

3. The expected output of those 3 queries are present in the data folder of the project2 Github repo. Filename: *sample_output.txt*
4. Till the final deadline, the auto-grader will execute the queries in point 2 against your index, and compare your result against the output in point 3. For the final submission, the query will be different, and will **not be shared** from beforehand.

16. I am getting a good score using the auto-grader, how do I make sure that my implementation yields correct results with other queries?

Create a dummy corpus of 10-15 documents on your own, along with a few dummy queries based on your corpus vocabulary. Work out the postings list, and queries & tf-idf scores by hand, and compare it against your implementation's output.