Airi Chow (#40003396)
COMP 479
Fall '20

Project #3

Using Python3
Yellow Highlight: Issue with running.
Green Highlight: Given names for outputs/inputs.

Subproject #1:

Sorting & Building an Index

Tested Commands

- **python3 spimi-indexer.py -i tokens.json**
- **Running -> build_index_block(args.input_file,args.output_file)**

==========
- Input: Lists of Tokens [ID, Token] (In this case of the entire REUTERS collection)
- Output: Sorted individual Blocks (with 500 unique terms and no duplicate IDs)
==========

os.chdir(os.getcwd() + "/BLOCKS"): The Block will be stored in this folder to not clutter up the root folder.

- The tokens are read line by line and parse to retrieve the docID then the term.
- If the term exists already in the dictionary, we have now to check if the current docID is already in the list of docIDs.
    > If that docID is a duplicate, it is ignored.
    > Else, we will append it to the list.
- If the term doesn't exist in the dictionary, we just add it to the dictionary as a key. The value is a list version of the docID.

- **len(current_dictionary) == 500**: After the # of keys in the dictionary reaches 500, we need to sort the dictionary by iterating through it and creating a sorted dictionary done as so: **sorted_dictionary = collections.OrderedDict(sorted(current_dictionary.items()))**
- This sorted_dictionary is then dumped into a file with the name "BLOCK" + block_counter.
- After this block has been created, the current_dictionary is clear and the block_counter goes up by one.

Merging the Blocks and creating the final index

Tested Commands

- **python3 spimi-indexer.py -i tokens.json**
- **Running -> create_final_index()**

==========

- Input: Block files
- Output: index.json which consists of a final merged index with sorted terms and docids (no duplicates)

==========

files = os.listdir(os.getcwd()): Used to account the # of blocks in the directory.
current_dictionary = dict(current_dictionary, **data): Data is a keyword parameter which is a JSON dictionary object that is passed to the current_dictonary and appended. For each block that is open, this is what will happen. In the end, current_dictionary will consists of all the terms from blocks.

sorted_dictionary = collections.OrderedDict(sorted(current_dictionary.items())): This has to be sorted once again before the final merging can be done.
os.chdir(".."): We change the directory back to root so that the intermediate "sorted_dictionary.json" can be written in the root folder. If not, it would interfere and mess up the "files" listed above and the count of blocks.

list(chain(*sorted_docID)): This is used to flatten the docID list into one list.

Timing:

```
Airis-MBP:Project3 Airi$ python3 spimi-indexer.py -i tokens.json
Total Time (Build Index Blocks): 26.96916675567627
2490
Total Time (Create the Final Index): 6.073323965072632
```

SPIMI total run time: 26.97 + 6.07 = **33.04 sec**
- Creating the blocks (500 terms each block): 26.97 sec
- Creating an index: 6.07 sec

```
Airis-MBP:Project2 Airi$ python3 ./block-5-sorter.py -i tokens.json
9519 12025 12797 12834 17483
Total Time (Build Naive Index): 31.765016078948975
```

- Creating an index using the naive way: 31 sec

It takes close to 5x as much time to create an index using the naive indexer. But SPIMI takes more time with creating the blocks. But creating the final index is much shorter.

Ranking:

- Incomplete. Ran out of time before deadline.

Querying:

Tested Commands

- **// Single Term //**
- **python3 query.py -i index.json -q hello**

==========

- Input: index.json (final merged index)
- Output: result.json consists of the query results

==========

Ranking wasn't implemented due to lack of time!

Single Term Querying

- If the query passed length is 1, then this would run.
- Upon finding the query term in the index dictionary, it would record the docID in a new dictionary called results and at the end write the results into results.json.
- If there are no results, it would print that the result cannot be found.

OR Querying

- for query_term in queries: Iteration for each query term.
- for term, docID in data.items(): Iteration to go through the index to find the matched query terms.
- if len(results) > 0: given the query term is found and a result has been added to the final dictionary consists of results, get the value.
- for single_docID in docID: for this iteration we append the unique docIDs to our list of docID and update the key's value.

- Finally, we sort the docID by transforming them to int so it sorts properly.
results[query] = [int(x) for x in results[query]]
results[query].sort()
(Note: with ranking, we would retrieve the top results so sorting would be done automatically based on the best results)

AND Querying

- It is not functioning properly.

if query_term == term and not query_term == "and":
// This is so we do not add the 'and' docID to our final docID list.

if len(results) > 0:
// Given we have a term in the dictionary, we need to compare with the next term's docIDs.

```
for single_docID in docID:
                    if single_docID in current_doc_id:
                       final_result.append(single_docID)
```

// The idea is that we will check the second's term docID with our current docID list. If the second's term docID exists in our current docID we will insert it into another dictionary(updated_doc_id). Then after iterating through the entire second's term docID, we would update the current docID with our updated_doc_id. Clear updated_doc_id. Then use the current docID to match with another term's docID.

Test Queries (4):

(A) Single Keyword Query:
python3 query.py -i index.json -q soft

{"soft": ["1", "7", "6", "288", "395", "472", "501", "1257", "1257", "1388", "1396", "1414", "1478", "1640", "2044", "2116", "2191", "2269", "2777", "3234", "3485", "3493", "3568", "3737", "3758", "3881", "3904", "4296", "4478", "4518", "4624", "4947", "5104", "5145", "6426", "6618", "6732", "6775", "6898", "6922", "6928", "7135", "7361", "7551", "8029", "9234", "9638", "9865", "10305", "10519", "10548", "10947", "10956", "11134", "11313", "11415", "11445", "11463", "12196", "12206", "12543", "12701", "12701", "12723", "12780", "13179", "13578", "14165", "14716", "14718", "14721", "14748", "14841", "14880", "14953", "15132", "15582", "15625", "15742", "15911", "15916", "15920", "16937", "16958", "17010", "17100", "17199", "17217", "17263", "17271", "17503", "17526", "17662", "17789", "18250", "18377", "18400", "18609", "18642", "18908", "19213", "19234", "19371", "19614", "19684", "19722", "19896", "19947", "19980", "21273"]}

- The results are similar to the naive indexer since soft wasn't part of the stopwords. But if there was Soft in the Project #2, this would have been added to the index. Unlike here where we didn't do any compression.

(B) BM25 was not implemented.

(C) AND Query
Not functioning properly.

(D) OR Query
python3 query.py -i index.json -q "moon soft"
Running OR query
Found Result for query term 'soft'!
{'moon soft': [1, 6, 7, 288, 395, 472, 501, 1257, 1257, 1388, 1396, 1414, 1478, 1640, 2044, 2116, 2191, 2269, 2777, 3234, 3485, 3493, 3568, 3737, 3758, 3881, 3904, 4296,

4478, 4518, 4624, 4947, 5104, 5145, 6426, 6618, 6732, 6775, 6898, 6922, 6928, 7135, 7361, 7551, 8029, 9234, 9638, 9865, 10305, 10519, 10548, 10947, 10956, 11134, 11313, 11415, 11445, 11463, 12196, 12206, 12543, 12701, 12701, 12723, 12780, 13179, 13578, 14165, 14716, 14718, 14721, 14748, 14841, 14880, 14953, 15132, 15582, 15625, 15742, 15911, 15916, 15920, 16937, 16958, 17010, 17100, 17199, 17217, 17263, 17271, 17503, 17526, 17662, 17789, 18250, 18377, 18400, 18609, 18642, 18908, 19213, 19234, 19371, 19614, 19684, 19722, 19896, 19947, 19980, 21273]}

Moon does not exist in the index but soft does so soft's docID are part of the result.

Sample Queries:

python3 query.py -i index.json -q "Democrats' welfare and healthcare reform policies"

AND not functioning properly.

python3 query.py -i index.json -q "George Bush"

{'George Bush': [2, 4, 5, 8, 43, 178, 255, 286, 342, 854, 871, 925, 925, 965, 1022, 1502, 1667, 1729, 2356, 2711, 2733, 2766, 2796, 2802, 3014, 3366, 3390, 3560, 3563, 3938, 4008, 4098, 4737, 4764, 4853, 5157, 5334, 5405, 5459, 5574, 5631, 5711, 6062, 6065, 6296, 6423, 6564, 6656, 6744, 6882, 6896, 6940, 6989, 7186, 7326, 7469, 7471, 7525, 7765, 8009, 8053, 8252, 8500, 8554, 8593, 8748, 8765, 9002, 9150, 9247, 9342, 9755, 10400, 10682, 10725, 11380, 11624, 12002, 12009, 12209, 12261, 12460, 12605, 12709, 12720, 12806, 12967, 13246, 13507, 14749, 14809, 14914, 15284, 15342, 15363, 15405, 15427, 15439, 15478, 15880, 16090, 16115, 16199, 16218, 16229, 16318, 16389, 16550, 16575, 16730, 16780, 16824, 17004, 17119, 17150, 17181, 17305, 17356, 17647, 18148, 18160, 18330, 18442, 18443, 18472, 18579, 18688, 18867, 18961, 19003, 19033, 19485, 19589, 19760, 19828, 19958, 20099, 20571, 20719, 20860, 20891, 21376, 21379, 21393]}

python3 query.py -i index.json -q "Drug company bankruptcies"

// Too long of a result because term frequency wasn't implemented and these terms span multiple documents. The printed results can be found in ***"result_drug_company_bankruptcies.json"***

Conclusion:

I learnt about how to implement SPIMI, and how to query with more than one single term. I didn't get the chance to take a look at ranking with BM25 due to the lack of time. But, I understand that it's an important aspect in trying to find information that is more

relevant to the query. If I had to redo the project, I think I would have start with Sub-project #2 so I made sure that I got the term frequency that I needed to apply the formula as well as the document frequency so I can use it for the OR querying.