Project 1
SPIMI Indexer

Submitted on: Monday, October 10th, 2017
Presented to: Dr. Sabine Bergler
COMP 479 / 6791 - Section SS
Fall 2017

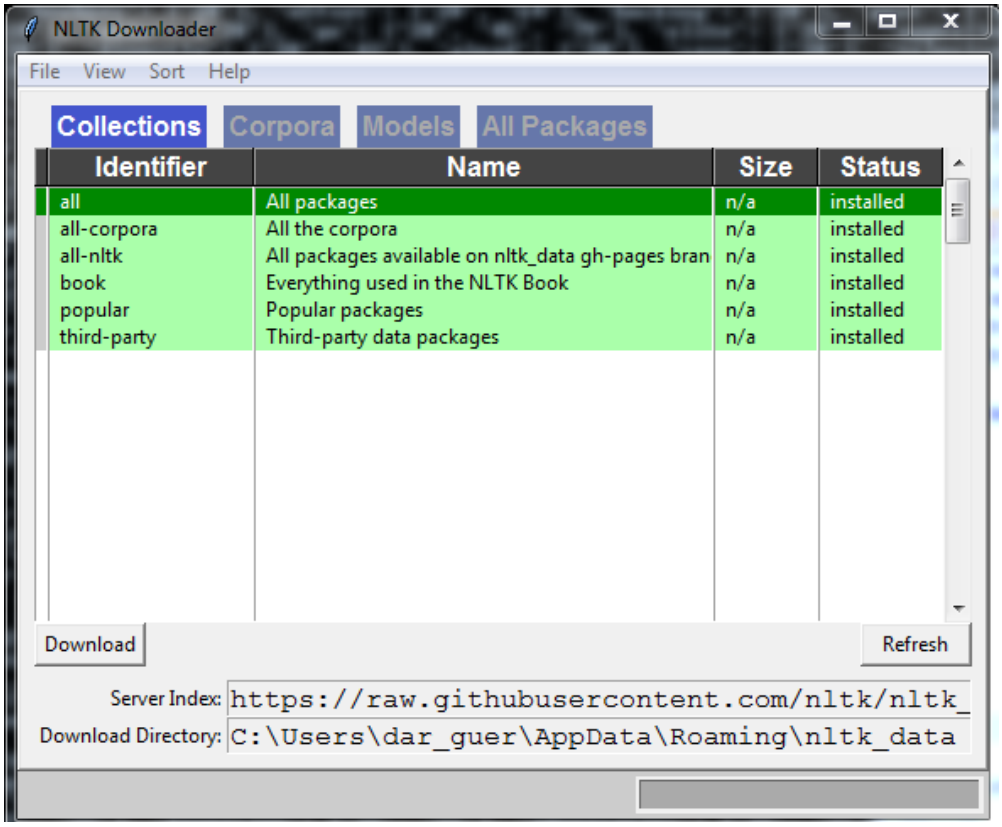
Submitted by: Darrel-Day Guerrero, 27352409

Environment Setup

Note that this project also utilizes virtualenv and can be installed through the following command: `python3 -m pip install -r requirements.txt`. However due to the restrictions on user permissions on Concordia's lab machines, please follow these steps in order for the project to be fully compatible:

1- Download and save the project folder to a directory path. (e.g. G:\comp479-p1)
2- Run Python 3.6 (32-bit) shell
3- Type and run the following commands to access the NLTK downloader

```
>>> import nltk
>>> nltk.download()
```



4- In the NLTK downloader, download and install all collections. From this point, all packages used in this project should be available. Note that beautifulsoup4's module is already included the project folder.

5- Open the command line and change into the project's directory.

```
cd G:\comp479-p1
```

6- To execute the project, type and run the following command with the following argument:

```
C:\Python36-32\python.exe main.py [block_size_limit]
```

where *block_size_limit* is an integer string argument

Approach

To build the SPIMI indexer, I first had to make sure that I understood certain concepts taught during my lectures: Boolean retrieval, document delineation, term vocabulary, postings lists, inverted index, and of course the single-pass-in-memory indexing (SPIMI) algorithm. As a result, Python was my language of choice due to its concise and compact characteristics when writing code and also due to well-known use in scientific and computational applications.

I designed this project with two main operations, each with their own set of modules:

1- Generate Inverted Index

Command

```
C:\Python36-32\python.exe generate_index.py [block_size_limit]
```

where *block_size_limit* is an integer string argument

generate_index.py

This file is the main entry point. With the help of other modules, this program calls the necessary functions to perform the following:

1- Parsing of all documents from the Reuters-21578 corpus
2- Preprocessing of each document by obtaining their tokens, compressing them and determining the set of terms.
3- Application of the single-pass in-memory indexing algorithm on each document and generate blocks that contain a term to postings list dictionary.
4- Merging of all SPIMI blocks into the final inverted index

reuters.py

This module is responsible for parsing all articles of the Reuters21578 collection. It utilizes the BeautifulSoup4 module and its HTML parser to extract content from the <TITLE> and <BODY> tags. As instructed, the documents are identified by their NEWID attribute which is extracted from the <REUTERS> tag.

vocabulary.py

This module preprocesses the data from all documents. It will first tokenize each document string. Then, it proceeds with compression and normalization. Here are some the design decisions implemented:

1- Discard tokens with punctuation marks (from Python's standard library string module)
2- Discard tokens with blanks or empty strings
3- Discard tokens with invalid code points from UTF-8
4- Apply lowercasing to all tokens
5- Discard tokens if a character is a digit

spimi.py

This module is responsible for applying the SPIMI algorithm to all of the documents. It has two main functions: one for inverting and another for the final merge of all SPIMI blocks.

The algorithm is very similar to the implementation in the book without the dynamical size adjustment for the posting lists and the main memory size limitation. In fact, my implementation uses arrays for postings lists and an artificial memory limitation for block sizes instead. The algorithm creates and writes a block dictionary {key: "term", value: [postings_list]} until the entire collection has been processed. Before writing a block into memory, it will sort the dictionary by alphabetically term order.

In final merge of all SPIMI blocks, an index/pointer to the first most term is created for each block (saved in a dictionary). Then, it determines the smallest (alphabetical) term posting list among all block pointers and extracts the term. If multiple pointers share the same term, then it combines all the postings and writes the {key: "term", value: [postings_list]} entry to the final index file (spimi_inverted_index.txt). The algorithm then moves the pointer(s) of block(s) with the last current term to the next line and repeats the previous steps until all lines from each block has been merged.

2- Query Document

Command

```
C:\Python36-32\python.exe query_doc.py
Enter your boolean query using && or || exclusively: [user_input]
```

query_doc.py

This file is main entry point for retrieving matching documents of user inputted queries. It parses the input to determine the type of query then executes the query (AND or OR).

Effect of preprocessing for Reuters

	Distinct Terms			Non-positional Postings		
	number	**Δ %**	**T %**	**number**	**Δ %**	**T %**
unfiltered	92271	N/A	N/A	1825487	N/A	N/A
no numbers	81594	-11.57%	-11.57%	1733379	-5.05%	-5.05%
case folding	65140	-20.16%	-29.40%	1615610	-6.79%	-11.50%
30 stop words	65112	~0%	-29.43%	1534196	-5.03%	-15.95%
150 stop words	65005	~0%	-29.54%	1284852	-16.25%	-29.61%
stemming	N/A	N/A	N/A	N/A	N/A	N/A

Positional postings have been omitted because of the nature of the inverted index created. Secondly, the algorithm the no number compression technique is specific. This explains why the compression is not very high. Basically, it evaluates string literals into the best matching Python literal, then it checks if the result is an instance of int or float. This means that tokens like “553,000”, “55th”, “0905” are not considered numbers.

These compression techniques were used to create a similar table from the book. To create my final inverted index and to retrieve matching documents, I took design decisions to compress my vocabulary even further (see vocabulary.py). For instance, according to the course slides, it is often good practice to lowercase everything since users will use lowercase regardless of correct capitalization.

```
print(" --- 1- Tokens with punctuation characters...")
print(" --- 2- Tokens with blank or empty strings...")
print(" --- 3- Tokens with invalid code points from encoding...")
```

```
    print(" --- 4- Tokens with digits or numbers...")
    print(" --- 5- Lowercase...")
```

Also, the stop words did not produce similar compression rates as in the Table because nltk always returns a random set of stop words. Therefore, it did not into consideration the most frequent stop words during its retrieval.

Other Design Ideas

While implementing the algorithm for retrieving matching documents, I noticed that there intersect() and union() set operations from Python's standard library. While I could have benefitted from simply using these functions, I decided to re-implement the intersect and union algorithms from the course material in order to develop my coding skills.

When determining the type of query inputted from the user, I use && (AND) and || (OR) characters so that I do not omit "and" and "or" keywords in my document retrieval module.

Test Queries

Single-Word:

```
>> audi
```

Result: [2362, 6481, 11026, 17474, 17570, 19436]

I wanted to search for articles about the German car manufacturer Audi. Despite the term being a formal name, it turns out the results were indeed relevant to the subject.

Multiple keyword query returning documents containing all keywords (AND):

```
>> nyse AND stock AND rise
```

Result: [205, 19207]

In this conjunctive query, I wanted to retrieve information about financial gains about the New York Stock exchange. I was satisfied with the result given the nature of my query terms. "NYSE" is known to be an acronym, "stock" has a high document frequency and "rise" can be also found in different contexts.

While the document 205 mentioned “the rise of computerized techniques”, 19027 certainly talked about a company stock rise and provided me with the information I wanted.

Multiple keyword query returning documents containing all keywords (OR):

```
>> turbo OR engine
```

Result: [264, 344, 627, 652, 727, 933, 970, 1132, 1252, 1497, 1685, 1864, 2718, 2735, 3069, 3185, 3202, 3350, 3398, 3457, 4406, 4568, 4724, 4755, 5022, 5521, 5825, 6318, 6462, 6535, 6811, 7671, 8365, 8404, 8716, 9353, 9403, 9458, 9662, 9771, 10502, 10740, 10747, 11247, 11404, 12034, 12460, 12816, 12894, 13049, 13772, 13773, 13774, 13865, 13966, 14031, 14314, 14402, 14748, 14878, 14961, 15014, 15263, 15286, 15314, 15337, 15501, 15698, 15764, 16557, 16618, 16659, 16691, 17403, 17696, 17797, 17799, 18288, 18659, 19289, 19436, 19637, 20764, 21507, 21523]

Straightforward query, I was searching for anything related to turbo or car engines. The results turned out to be sparse for “turbo” but relevant for “engine”.

Sample Queries

```
>> Jimmy && Carter
```

Result: [12136, 13540, 17023, 18005, 19432, 20614]

```
>> Green && Party
```

Result: [21577]

```
>> Innovations && in && Telecommunications
```

Result: None

Query Exchange

Single Word 1:

Single Word 2:

Single Word 3:

Multiple AND 1:

Multiple AND 2:

Multiple AND 3:

Multiple OR 1:

Multiple OR 2:

Multiple OR 3:

Summary

I enjoyed working this assignment. Working with a large (but fairly small and outdated with respect to today's data), outdated corpus allowed me to understand the fundamentals of basic Boolean retrieval and indexing. From this point, one can only imagine how there are many other variables in play when considering working with even larger and more complex sets of collections out there. This project was only the tip of the iceberg as we did not dive into positional indexing, spelling corrections, different languages and formats, etc. However, I am glad to have created a simple working SPIMI indexer. Also, I was impressed with the resources Python offers to perform set operations, list manipulations, etc.