Konrad Rauscher
Introduction to Information Retrieval
Project 1
10/7/16

Status:
Complete, except for date validation and complete date normalization

Approx. Time Spent on Project:

85 hours

Things I wish I had been told prior to being given assignment:
NA

Instructions:
set parameters on the following lines:
setting file path: line 881
setting memory constraint: line 879
selecting index: 883-889
selecting size of phrases for phrase index: 890

Design:
Sources: *nltk.corpus.stopwords, py2casefold, BeautifulSoup, numpy, glob, re,*

The following steps are executed on a per-data-file basis, using glob.

(Same for all indexes, except stop words are removed for all indexes except for positional)

Preprocessing & Tokenization:
1) use BeautifulSoup to parse the html format of the file into a soup object
2) using a regular expression, remove unwanted doc tags
3) using my soup object, find all text sections, and remove comments and:
    1) use regular expressions to substitute special symbols, and remove unwanted unicode characters
    2) take care of hyphenated terms: keep terms w/ common prefixes (pre, post, re, etc.) as both 'preprocessing' AND 'processing'
        1) up to three hyphens --> part-of-speach --> 'part', 'speach', 'partofspeach'
    3) Convert alpha digit (CDC-50) to cdc50 and cdc (storing as separate terms if more than three letters)
    4) Convert digit alpha (50-CDC) to 50cdc and cdc (storing as separate terms if more than three letters)
    5) normalize digit formats (100.00, 1,000.00 and 1,000 —> 1000)
    6) normalize MM/DD/yyyy to MM-DD-YYYY
    7) perform case folding
    8) remove stop words if remove stop words parameter is True.
4) return a dictionary with doc IDs as keys and the processed text corresponding to that doc ID

<u>Single term index</u>

Conversion to triples:
1) for each doc number and corresponding document text in the document:
    1) create a list of triples (in the form of *term, doc id, freq*) for all the terms in a given document text
    2) check the size of the buffer of triples is less than the memory constraint. If this condition is not met, then the the buffer of triples is flushed to a temp file on the disk, and the triple list is added to the just emptied buffer. If this condition is met, merge the list of triples from the current document into the buffer, incrementing frequencies for terms already existing in the buffer, and adding entire triples for terms that don't exist.
2) If there are still elements in the buffer after the above loops finishes, flush the remaining triples to the disk in a temp file

Merge:
1) iterate through all temp disk files:

    1) convert the first line (so the smallest in terms of alphabetical order) from each temp disk file from a string to a triple, append the file number from which the triple comes from (to be used to replace triples that are popped of the min triple list later) ad add to list of min triples (effectively a min heap).
2) while list of min triples is not empty:
    1) sort list of min triples (to account for previous additions to the list)
    2) store the file number of the triple at the ‘top’ of the ‘heap’
    3) delete the 4th index of the same ‘triple’ (containing this file number)
    4) convert this triple to a string and output to the index file, and pop this triple of the min heap
    5) load a new triple into the min heap, from the same file as the triple that was just written to the index file and popped of the heap, if this file still has triples that have not been read. Otherwise, don’t add a triple for this iteration and the size of the list of min triples will decrease by one.

<u>Stem index</u>

Conversion to triples:

1) for each doc number and corresponding document text in the document:
    1) user porter stemmer to update the document text to stems of the original terms
    2) create a list of triples, but this time with an additional index containing a list of all positions of each occurrence (position in terms of location in the list of tokens for a given document) (in the form of *term, doc id, freq, [pos0, pos1, etc.]*) for all the terms in a given document text
    3) check the size of the buffer of triples is less than the memory constraint. If this condition is not met, then the the buffer of triples is flushed to a temp file on the disk, and the triple list is added to the just emptied buffer. If this condition is met, merge the list of triples from the current document into the buffer, incrementing frequencies for terms already existing in the buffer, appending positions to triples of the same term, and adding entire triples for terms that don’t exist.
2) If there are still elements in the buffer after the above loops finishes, flush the remaining triples to the disk in a temp file

Merge:

1) iterate through all temp disk files:
    1) convert the first line (so the smallest in terms of alphabetical order) from each temp disk file from a string to a triple, append the file number from which the triple comes from (to be used to replace triples that are popped of the min triple list later) ad add to list of min triples (effectively a min heap).
2) while list of min triples is not empty:
    1) sort list of min triples (to account for previous additions to the list)
    2) store the file number of the triple at the ‘top’ of the ‘heap’
    3) delete the 4th index of the same ‘triple’ (containing this file number)
    4) convert this triple to a string and output to the index file, and pop this triple of the min heap
    5) load a new triple into the min heap, from the same file as the triple that was just written to the index file and popped of the heap, if this file still has triples that have not been

read. Otherwise, don't add a triple for this iteration and the size of the list of min triples will decrease by one.

<u>Phrase index</u>

Conversion to triples:

1) for each doc number and corresponding document text in the document:
    1) if the size of the phrases to be captured is 2:
        1) iterate through the each token for the current document, combing the i th and i + 1 th indices and appending them to a temp list, which the docText is updated to, once the loop condition is no longer met (i being less than one less than the length of the doc text)
    2) else if the size of the phrases to be captured is 2:
        1) iterate through the each token for the current document, combing the i th, i + 1 th, and i + 2 th indices and appending them to a temp list, which the docText is updated to, once the loop condition is no longer met (i being less than two less than the length of the doc text)
    3) create a list of triples, but this time with an additional index containing a list of all positions of each occurrence (position in terms of location in the list of tokens for a given document) (in the form of *term, doc id, freq, [pos0, pos1, etc.]*) for all the terms in a given document text
    4) check the size of the buffer of triples is less than the memory constraint. If this condition is not met, then the the buffer of triples is flushed to a temp file on the disk, and the triple list is added to the just emptied buffer. If this condition is met, merge the list of triples from the current document into the buffer, incrementing frequencies for terms already existing in the buffer, appending positions to triples of the same term, and adding entire triples for terms that don't exist.
2) If there are still elements in the buffer after the above loops finishes, flush the remaining triples to the disk in a temp file

Merge:

1) iterate through all temp disk files:
    1) convert the first line (so the smallest in terms of alphabetical order) from each temp disk file from a string to a triple, append the file number from which the triple comes from (to be used to replace triples that are popped of the min triple list later) ad add to list of min triples (effectively a min heap).
2) while list of min triples is not empty:
    1) sort list of min triples (to account for previous additions to the list)
    2) store the file number of the triple at the 'top' of the 'heap'
    3) delete the 4th index of the same 'triple' (containing this file number)
    4) convert this triple to a string and output to the index file, and pop this triple of the min heap
    5) load a new triple into the min heap, from the same file as the triple that was just written to the index file and popped of the heap, if this file still has triples that have not been read. Otherwise, don't add a triple for this iteration and the size of the list of min triples will decrease by one.

Single term positional index

Conversion to triples:

1) for each doc number and corresponding document text in the document:
    1) create a list of triples, but this time with an additional index containing a list of all positions of each occurrence (position in terms of location in the list of tokens for a given document) (in the form of *term, doc id, freq, [pos0, pos1, etc.]*) for all the terms in a given document text
    2) check the size of the buffer of triples is less than the memory constraint. If this condition is not met, then the the buffer of triples is flushed to a temp file on the disk, and the triple list is added to the just emptied buffer. If this condition is met, merge the list of triples from the current document into the buffer, incrementing frequencies for terms already existing in the buffer, appending positions to triples of the same term, and adding entire triples for terms that don't exist.
2) If there are still elements in the buffer after the above loops finishes, flush the remaining triples to the disk in a temp file

Merge:

1) iterate through all temp disk files:
    1) convert the first line (so the smallest in terms of alphabetical order) from each temp disk file from a string to a triple, append the file number from which the triple comes from (to be used to replace triples that are popped of the min triple list later) ad add to list of min triples (effectively a min heap).
2) while list of min triples is not empty:
    1) sort list of min triples (to account for previous additions to the list)
    2) store the file number of the triple at the 'top' of the 'heap'
    3) delete the 4th index of the same 'triple' (containing this file number)
    4) convert this triple to a string and output to the index file, and pop this triple of the min heap
    5) load a new triple into the min heap, from the same file as the triple that was just written to the index file and popped of the heap, if this file still has triples that have not been read. Otherwise, don't add a triple for this iteration and the size of the list of min triples will decrease by one.

Construct Lexicon (same for all indexes):

1) Outputs lexicon to a file, that contains all terms (in alphabetical order) with their document frequencies, and total # of documents at end of file
2) outputs the following index statistics: mean df, median df, max df, min df, number of terms in lexicon.

Design Decisions and Subsequent Implications:

Decision: use terms directly instead of term IDs
pros: simplicity, implementation time.
cons: had to use hash tables to increase efficiency in look ups. Still less efficient than using term IDs however, as more computationally expensive to compare as well as find strings than ids.

Decision: m-way merge instead of 2-way merge
pros: quicker merge.
cons: took quite a while to implement; required lots of debugging

Decision: how indexes are stored:
1) the posting list is stored in a file as a list of triples (might involve more than three terms for some indexes) in alphabetical order by term and doc ID, where each term in the triple is separated by a ','
2) the lexicon containing is stored in a separate file as a list of terms in the form of (*term*, *df*)
pros: easy to implement and parse. human-readable.

Decision: implementation of memory management:
1) when creating triples for each term and saving to temp disk files: used a buffer whose contents were flushed to a temp disk files if its size became larger the the memory constraint
pros: effective
2) When merging temp disk files:
*cons: T*he number of elements in the min heap (list_minTriples) (determined by the number of temp disk files being merged from) can conceivably become greater than the memory constraint. In all of the merges for the different indexes in this project, the size of list_minTriples never exceed 128, so I did not implement a mechanism for flush the min heap to the disk.

Decision: library usage
1) nltk.corpus
pros: provides an adequate list of stop words
2) py2casefold
pros: straightforward way to perform a case fold on text
3) BeautifulSoup
pros: allows for simple interfacing with the html format of the data files (finding all text sections, etc.)
4) numpy
pros: allows for easy calculations of medians, minimums, maximums, means, etc.
5) glob
pros: allows for iteration through all data files in the same directory
6) re
pros: provides regular expression functionality to capture and identify specific text patterns, useful for preprocessing.

Sources: *nltk.tokenize, nltk.corpus.stopwords, py2casefold, BeautifulSoup, numpy, glob, collections, re*

Analysis:

Report 1

	Lexicon (# of terms)	Index size Lexicon+PL(byte)	Max df	Min df	Mean df	Median df
Single term index	3740	45 kb + 188 KB	11	1	1.6	1.0
Stem Index	2689	29 kb + 142 kb	11	1	1.85	1.0
Phrase Index	109543	754 kb + 1.6 MB	67	1	1.27	1.0
Single term positional index	2857	33 kb + 3.3 MB	11	1	1.65	1.0

First, in terms of the number of terms contained in each lexicon, it makes sense that the stem posting list and lexicon would have a smaller byte size, as stemming reduces words with the same stem to a single stem, so there would be fewer terms in both the lexicon and posting list.

Regarding the index size for each index, it makes sense that both the size of the lexicon and posting list for the phrase index would be distinctly larger as phrases involve terms that much larger, and thus take up more space.

With max df, 3 of the 4 indexes have a value of 11. This seems odd as I would expect the single term positional index which does not remove stop words that have a distinctly higher max df, as a stop word such as 'the' or 'a' would appear in almost all documents. After checking however, 'the' is actually at least one of the terms with a *df* of 11, so I suppose this makes sense.

All indexes have a min *df* of one, which makes sense I as am not executing any pruning to remove low frequency terms and only a stop word list to remove high frequencies terms (except for single term positional, which does not remove stop words).

Looking at mean *df,* the two outliers are stem with a 1.85 and and phrase with a 1.27. This makes sense as a phrase index has terms that are constructed out of adjacency in a given document, thus introducing a differentiation to terms that would otherwise be similar and the stem index reduces similar terms (in terms of stems) to the same term, thus converging the occurrences of these similar terms to a single term, increasing *df* as a whole.

In terms of the median *df* for all indexes bing one, this again makes sense because I am not executing any pruning to remove low frequency terms and only a stop word list to remove high frequencies terms (except for single term positional, which does not remove stop words).

Report 2

<u>Single</u>

	1000	10000	100000	Unlimited Memory
Time taken to create temporary files (up to merging)	230.4	228.297599	223.935554	227.447933
Time taken to merge temp files	0.7	0.666285	0.681245	0.640833
Time taken to build Inverted Index in milliseconds (whole process from reading documents to building inverted index)	227	228.963887	224.616802	228.088768

<u>Positional</u>

	1000	10000	100000	Unlimited Memory
Time taken to create temporary files (up to merging)	320.25091	312.817688	303.868115	303.394091
Time taken to merge temp files	1.699345	1.68361	1.699119	1.779708
Time taken to build Inverted Index in milliseconds (whole process from reading documents to building inverted index)	358.91318	314.501292	305.567232	305.173796

<u>Stem</u>

	1000	10000	100000	Unlimited Memory

Time taken to create temporary files (up to merging)	226.452182	218.949636	212.863103	210.945149
Time taken to merge temp files	0.453461	0.495891	0.453268	0.468435
Time taken to build Inverted Index in milliseconds (whole process from reading documents to building inverted index)	226.905648	219.445535	213.316375	223.413586

Phrase (2 term)

	1000	10000	100000	Unlimited Memory
Time taken to create temporary files (up to merging)	577.964899	535.174866	501.736135	500.855466
Time taken to merge temp files	29.648028	28.562978	28.132537	27.761874
Time taken to build Inverted Index in milliseconds (whole process from reading documents to building inverted index)	407.612931	563.737849	529.868678	544.617344

Regarding the time taken to create temporary files (up to merging), the value of this metric generally decreases across all indexes as the as the size of the memory constraint is increased. Because the factor being affected by the change in the memory constraint is the number of temp disk files being created, this observation seems to indicate that the creation of and interaction with a greater number temp disk files is more costly in terms of time.

Time taken to merge files: single and stem took much less time than positional and phrase. This makes sense as there is more information being merged in the case of these later two indexes.

The total time taken to the index for single and stem is again much less time intensive than it is for positional and phrase. This makes sense as there is more information being merged in the case of these later two indexes.