Christopher Zawora
Information Retrieval
10/15/15

Project 1 - Pre-processing and Indexing

Cover page

Project: complete

Hours worked: 40-45 hours

Things I would like to have known:

It would have been more useful to go over efficient use of regex and how text normalization and tokenizing is done with it. The slides from class did not give many in depth examples of using regex

Design Document

The general plan for this project is to successively open the files and loop through line by line looking for <DOC> and <TEXT> tags. All the lines of a specific document along with the document ID will be stored together in unicode encoding in a Doc object.

Each Doc object will then be put through the normalization process. This case folds all the letters to lowercase, replaces all the useful &**** strings with their unicode equivalents. Then all the dates are found and normalized into the mm/dd/yyyy format. This will still allow for partial date lookups since all other '/' characters will be removed. So a month from a date can be searched with 'mm/' and a year can be searched with '/yyyy'.

After dates have been normalized then the document will be tokenized. For all but the phrase index the tokenization will first capture IP addresses, URL's, emails, standardize numerals, and capture monetary values. It will then deal with abbreviations and split the text on all forbidden characters. Finally, it will deal with the hyphenated words. However, for phrases the goal is to keep words together that are not separated, so all special tokens and characters will be replaced with a "<" char. That way one final split can be done by '<' and this will leave you most of the phrases. Then all that is left to check for phrases is the stop words and length of the resulting splits.

The lists of tokens will be passed and a document frequency for that token will be calculated. The token will be added to lexicon if it does not exist. Then depending on the memory requirements, the token/count pair will be immediately put into the inverted index or put into a collection of posting list tuples.

If operating under memory constraints, after every document is processed the count of posting list tuples will be checked and if it exceeds the limit, the tuples will be written to a temp file. Once all documents are processed, the tuple will be merged in groups of 25 until there are less than 25 temp files left. The merged temp files will then be finally merged and written to an inverted index on disk

If there are no memory constraints, then when all documents are processed the inverted index in memory will be written to disk.

This flow will be implemented through 3 classes: Doc, Index, and IndexMachine. The Doc class will keep the document ID and the document text encapsulated. The Index class will be where the inverted index, the posting list tuples, and lexicon will be stored. The IndexMachine class acts as a wrapper class that will handle all the processing of the documents and creation of the index. The main function will only gather the inputs from the command line regarding index type and memory requirements and then create an instance of IndexMachine and begin it.

I plan to use Python for this project. One main reason for this is due to its advantages when it comes to text and string processing. The other reason is that I foresee its dictionary data structure being useful in quickening lexicon insertions and searches as well for creating the inverted index. Its map function combined with lambda functions might also be useful.

Flow diagram:

IndexMachine object is created with index and memory options —> Trec file opened —> individual document brought into memory and Doc object created —> document is normalized and split into tokens —> tokens are then added to the inverted index or kept in a list to be written to a temp file in the Index object—> after the documents are processed merge temp files if necessary and write inverted index to disk

Analysis

Tables

1 - Index Statistics

	lexicon #	index size (lexicon + PL)	max df	min df	mean df	median df
single term index	38891	779 KB +7.2 MB	1099	1	9	1
single term positional	39361	827 KB + 16.8 MB	1742	1	12	1
phrase index	123000	3.8 MB +4.9MB	564	1	1	1
stem index	31183	596KB +6.3MB	1143	1	10	1

The index statistics show that most of the lexicons' sizes were roughly there same. It makes sense that the single term positional index is slightly bigger than the single term index since the single term positional index included stop words. The single term positional index also gets much larger when you look at its posting list, due to the fact that it is also storing the positions of occurrence of each token. The stem index is slightly smaller than the single term index probably due to the fact that the stemming process folded some similar words together. The outlier is the phrase index lexicon which is 4 times the size of the other indices' lexicons. This also makes sense given that it is made up of a combinations of the words in the other indices. With more combinations, the lexicon grows, as seen in how much memory the lexicon uses.

The document frequencies occurring in the indices are also reasonable. The phrase index has a lower max df due to the fact that there are more unique phrases that are not recurring that often. Meanwhile, the single term positional index has a much high max df which is most likely caused by the inclusion of stop words in the index. The min df is 1 for each index which is expected. However, the median df in each index being 1 is more interesting a result since it shows that a large number of the tokens in the index are only appearing once in the corpus. Lastly, the mean df shows that the phrase index is made up of tokens that on average are appearing only once. The mean df for the stem index is slightly higher than the mean df of the single term index, another result of stemming. Additionally, the mean df of the single term positional index also benefits from the inclusion of stop words in the index.

2 - Index running times

single term index

	1000 tuples	**10,000 tuples**	**100,000 tuples**	**unlimited**
time to up completion of temp files	523s	527s	521s	650s
time to merge temp files	34s	9s	2s	-
time to make completed inverted index	560s	536s	525s	650s

single term positional index

	1000 tuples	**10,000 tuples**	**100,000 tuples**	**unlimited**
time to up completion of temp files	533s	538s	604s	767s
time to merge temp files	50s	14s	6s	-
time to make completed inverted index	583s	552s	610s	770s

stem index

	1000 tuples	10,000 tuples	100,000 tuples	unlimited
time to up completion of temp files	524s	524s	538s	636s
time to merge temp files	25s	11s	3s	-
time to make completed inverted index	550s	535s	541s	638s

phrase index

	1000 tuples	10,000 tuples	100,000 tuples	unlimited
time to up completion of temp files	749s	753s	697s	1074s
time to merge temp files	46s	6s	2s	-
time to make completed inverted index	796s	759s	699s	1074s

There are a couple trends present across the running time data of the different indices. One trend is that as the memory constraint decreases and more tuples are allowed in memory, the time to create the temp file increases or stays the same. However, the more tuples in each temp file the quicker they are merged. In the phrase index and the single term index, these rates of change lead to a shorter total time to create the inverted index. On the other hand, in the single term positional index and the stem index the time to make the temp files grows quicker than the time to merge the files decreases, leading to an overall increase in time.

The other trend in the data is that all the indices performing under memory constraints end creating a completed inverted index quicker than the indices with unlimited memory. This is a striking result and counterintuitive to expectations. One would expect that the creation and merging of temp files would slow the down the process and the indices with unlimited memory would skip this lag time. However, the opposite is true. After a re-examination of my code, I realize this result is due to my implementation of unlimited memory indices. When the temp files are created, they are written to disk in sorted order. Then when merging the files the ordered format of the posting list tuples allows for a quick packaging of the inverted index. But when I create the inverted index in memory I do so using a dictionary data structure taking in the data in an unsorted manner. Due to this algorithm, I am forced to check the keys of the dictionary every time I insert a new tuple to make sure that the term ID already exists in the dictionary. Furthermore, as the inverted index, or the dictionary, grows, the key list becomes very large and

searching through it constantly is sure to cause lag time. A potential solution to this problem would be process all the documents first and create a large list of all the posting list tuples. Then sort the tuples at this point and from the sorted tuple list, quickly be able to loop through it once and produce a finished inverted index on disk.