CIS*4500 (Fall 2019) Machine Learning for Natural Language Processing

Assignment Two

Due on Sunday, Nov. 3, 2019 by 11:59 pm

General Description

An *information retrieval* (IR) system typically involves three steps: (1) data preprocessing; (2) offline processing; and (3) online processing. In data preprocessing, we tokenize text into a sequence of tokens and then filter out those irrelevant ones. The scanner we developed in Assignment One can help us do the tokenization. After that, we can filter out certain kinds of tokens such as punctuations and numbers, remove stop words, and convert words to their stems. The offline processing, also called indexing, takes the preprocessed documents as input and generate the inverted files as output. Since the document collections are getting increasingly bigger these days with the web being the extreme, this step usually takes a long time to complete, and as a result, it is only updated periodically to refresh the index. The online processing follows a particular IR model, but most of the IR models can be implemented efficiently with the index created from the offline processing. This step should be interactive, allowing the user to enter one query at a time and get a list of matched documents in the order of relevancy.

Creating the Preprocessed Documents

For this assignment, a dataset named `documents.txt` is provided in the "A2-Data-and-Samples" package on CourseLink. Using the first two programs you developed in Assignment One, we can generate the output file `documents.tokenized`. After that, you should write another program that adds the following tasks for further preprocessing:

(1) Normalization: convert all token values into lower cases so that different mixes of the cases for the same word can be matched (e.g., Book, book, and BOOK are converted to the same word "book").

(2) Further Filtering: remove certain kinds of tokens, including all numbers and punctuation marks, since they are unlikely to be used in a query for information retrieval.

(3) Stop Word Removal: a list of manually built stop words for English (e.g., "the", "a", "in", and "of") will be provided as the file `stopwords.txt` on CourseLink. By looking up this list, any matched words in the input can be removed.

(4) Stemming: convert words to their stems. For example, "computer", "computers", "compute", "computed", "computing", "computation", and "computational" can all be reduced to the stem "comput". We can use the stemmer in OpenNLP to do the task (see the sample program "StemmerDemo.java" on CourseLink for an example).

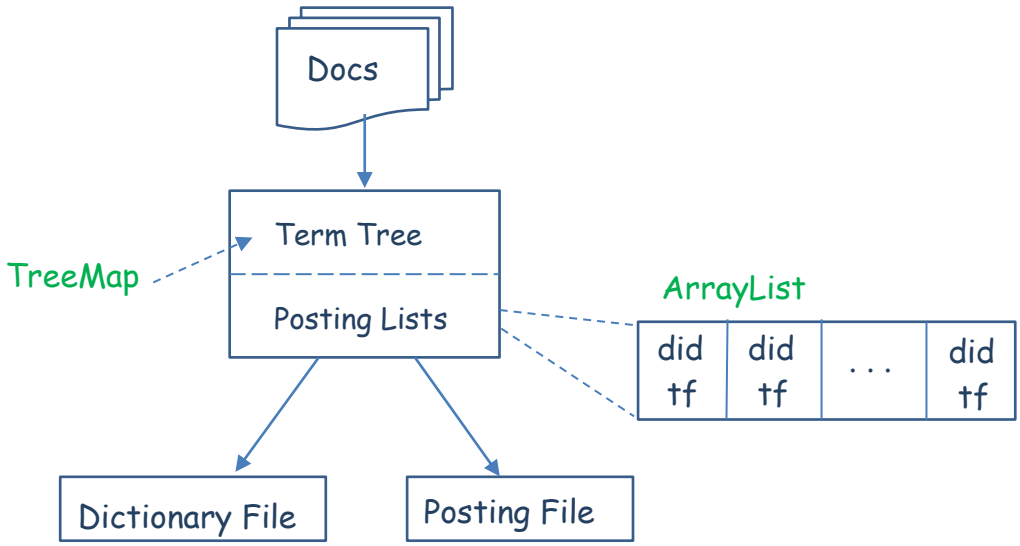
Please note that the above tasks should only be applied to the sentences of the $TITLE and $TEXT sections of each document, and the output file should have the same structure as the input file except that only the normalized stems are kept for all documents.

Offline Processing

For the offline processing, we take the preprocessed file as input, and produce an inverted index in terms of three output files: `dictionary.txt`, `postings.txt`, and `docids.txt`. As illustrated in the following figure, we need to read the given documents sequentially and maintain all unique stems in a TreeMap, and associate each stem with an ArrayList of posting entries, each of which contains a relative document number (denoted as `did`) and the term frequency (denoted as `tf`) of how many times the corresponding stem appears in the given document. Every time we get a stem from the input file, we will handle it in one of the three situations.

```
if the stem is not on the TreeMap:
   add the stem to the TreeMap and associate it with
      a new ArrayList
   add a new posting entry to the ArrayList with did
      set to the current document number and tf set to 1
 else if the stem is on the TreeMap:
   get the associated ArrayList for the stem
   if its last entry matches the current document number:
      increase its tf by 1
   else
      add a new posting entry to the ArrayList
      set did to the current document number and tf to 1
```

We choose TreeMap and ArrayList for our implementation because we need dynamic structures to maintain all unique stems and their corresponding positing entries since we do not know how many stems and posting entries are there until we finish the processing of the entire input file. Another reason of using TreeMap is that we can read off all the stems in an alphabetic order, which is desirable for generating the `dictionary.txt` file. Note that the relative document number starts from 0 and increments as we read the documents one by one from the input file. Along with the TreeMap and ArrayList's, we also need to maintain another ArrayList that records the docids (e.g., LA010190-0001), the titles, and the starting line numbers for each document in the input file.



After all the documents are processed, we can generate three output files from these dynamic structures. File `dictionary.txt` contains all the stems in the TreeMap along with the number of documents in which they occur. These stems will be stored in the alphabetical order based on the following format:

<total number of stems>
<stem1> <document-frequency1>
<stem2> <document-frequency2>
….

File `postings.txt` is the concatenation of all the contents in the ArrayList's and is ordered according to the corresponding stems in `dictionary.txt` with the following format:

<total number of entries>
<did1> <tf1>
<did2> <tf2>
…

File `docids.txt` is the set of docid's along with their titles and starting line number in the input file and is structured as follows:

<total number of documents>
<docid1> <start-position1> <title1>
<docid2> <start-position2> <title2>
…

Such a structure allows us to display the search results for each query later on during the online processing. On CourseLink, we also provide a sample program named "TreeMapDemo.java" to show some of the useful methods for offline processing.

Online Processing

For the online processing, we need to load the inverted files generated from the offline processing into memory so that they can be accessed efficiently. Since we now know the total number of stems in `dictionary.txt` and the total number of entries in `postings.txt`, we can store them into two static arrays so that the "dictionary" array becomes a list of stems along with their offsets (i.e., the starting positions of the related posting entries) in the "postings" array, which is a list of posting entries made of did and the term frequency in that document. Note that for each stem in `dictionary.txt`, we have a document frequency for it, indicating in how many documents the stem appears in the dataset. In "dictionary" array, we need to change the document frequency to an offset that points to the starting position of the stem's posting entries in the "postings" array. To do this, we can add all the document frequencies processed so far to get the offset to the "postings" array for the next new stem. Later, if we need the document frequency of this stem, we can simply compute the difference between the offsets of the current stem and the next consecutive stem. For example, if the offset for stem "abc" is 100 and the stem has a document frequency of 5, the offset for the next consecutive stem will be (100 + 5) = 105. Later on, we can simply do the subtraction of (105 – 100) to get the document frequency of 5 for the stem "abc".

Also note that we use the relative document number "did" in the "postings" array. To get the corresponding "docid" (e.g., LA010190-0001), we need to load the `docids.txt` into another static array with the same structure. Then, using "did" as the index, we can retrieve the corresponding "docid" along with its title and starting line number in the input file.

Once these files are all loaded into the memory, the system can now enter a command loop where we can take a query at a time and return the search results to the user, or enter "q" or "quit" to terminate the loop and the online processing system. In displaying the search results, we simply show the similarity values, the docid's, and the titles for up to 10 top-matched documents in the dataset.

To search for the relevant documents of a given query, we need to implement an IR model, and for this assignment, you will focus on the vector space model and use the inner products to compute the similarity values. Please refer to the lecture notes for the detailed formulas to compute the weights and similarity values. In addition, all the words in a query should be preprocessed in order to match the stems in the inverted index. However, since queries are typically short (two to three words on average for the web search), we can simply split them by white spaces, and then use the methods defined in the data preprocessing step to do case normalization, further filtering, stop word removal, and stemming before they are used to search for the relevant documents.

For testing, you can enter any query you like, or choose ones from a standard test set for this data collection, called `queries.txt` on CourseLink. Due to the time constraints, we will not perform any formal evaluations in terms of Precision, Recall, and F-measures in this assignment, but for a research paper, such evaluations are normally conducted to demonstrate the performance of the implemented IR system.

Makefile

You are responsible for writing a Makefile to maintain all of your Java code. Specifically, typing "make" or "make all" should result in the compilation of all three required programs: i.e., `Preprocessor` for further data preprocessing, `Indexer` for offline processing, and `Retriever` for online processing. You should ensure that all required dependencies are specified correctly.

Guidelines for Assignment Submission:

Same as in Assignment One except that you need to organize all of your files into a directory named by the pattern "`userid_a2`" (e.g., "`fsong_a2`") and hand in a compressed file in the related dropbox on CourseLink before the due time.