HW3: Inverted-index creation

Summary

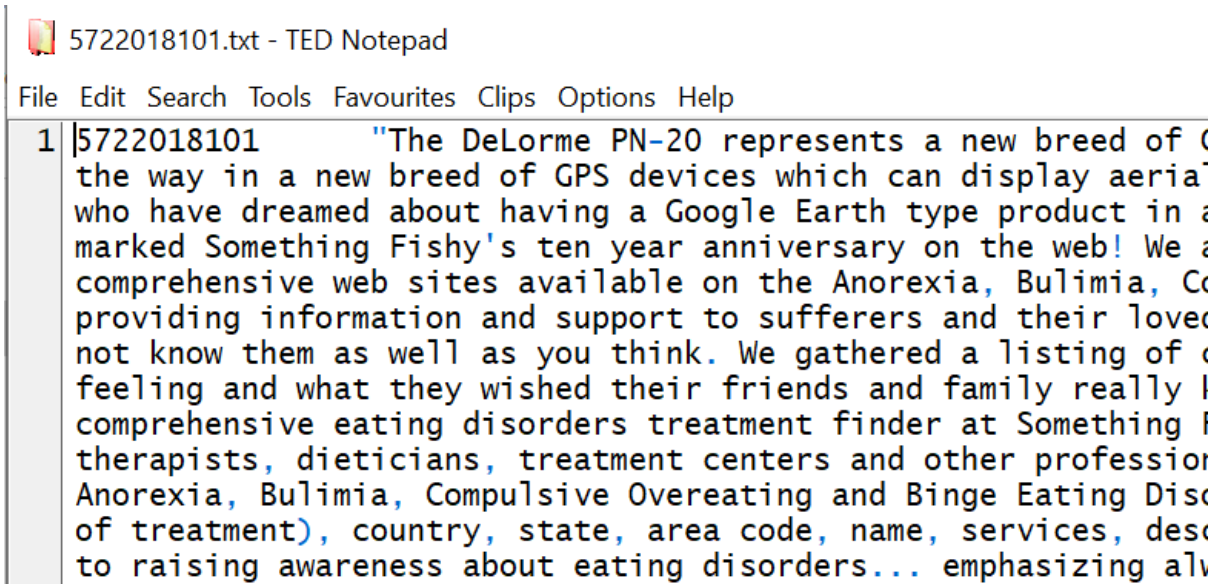
In this homework, you'll write code that indexes words from multiple text files, and outputs an inverted index that looks like this:

```
answer  5722018453:2 5722018483:1
antecedence 5722018502:1 5722018435:1
asterisks.  5722018417:1 5722018504:2 5722018447:1
beautiful   5722018439:7 5722018417:2 5722018416:3 5722018438:5 5722018437:1 5722018415:1 5722018414:2 5722018435:3
bind    5722018419:6 5722018417:39 5722018416:1
chunking    5722018507:1 5722018502:1
```

Description

You will need a collection of text files to index, here they are. After you unzip it, you'll get two directories with text files in them: 'devdata' (with 5 files), 'fulldata' (with 74 files).

The input data is already cleaned, that is all the \n\r characters are removed - but one or more \t chars might still be present (which needs to be handled). There is punctuation, and you are required to handle this in your code: replace all the occurrences of special (punctuation) characters and numerals with the space character, and convert all the words to lowercase. A single '\t' separates the key (docID) from the value (actual document contents). In other words, the input files are in a key-value format where docID is the key, and the contents are the value; the key and value are separated by a tab:



The above format is to help you build the inverted index easily - the filename (docID) is in the file's text so you can simply extract it.

As you know, Google invented MapReduce, to help them do this on a massive scale. For this HW, we could have based it on GCP/Azure/AWS/... where you would upload the files to the cloud and launch MapReduce jobs there. To get practice doing this, please do try it own your own, after this course.

But for this HW, there is a much simpler way! It's this [contains code I got from elsewhere, and the custom environment I set up to run it (ie. the minimal collection of .jar files needed, including the

one for MapReduce)]: **start with https://replit.com/@satychary/HadoopWordCounter** - it is a self-contained, complete, minimal MapReduce example that counts words in two input documents. Study it thoroughly - see what files are used (code, data, config), how the code is organized (a single class called WordCount), how it is run (we specify input and output folders). Be sure to look at the three columns: files on the left, file contents in the center, execution on the right. **Fork it, to do your homework - in other words, do it on your own repl.it area** (you need to sign up for a free account). FYI, repl.it is built on top of GCP: https://cloud.google.com/customers/repl-it [how cool!].

For the HW, you need to create a unigram index, and a bigram one. Details are in the two sections that follow.

Unigram index

You will need to create a file called unigram_index.txt, containing words from files in fulldata.

Modify the mapper in the repl.it link above , to **output (word, docID:count)**, instead of what it currently outputs for word counting, which is (word, count); also, **use a HashMap** data structure in your reducer.

Bigram index

You'll create a file called selected_bigram_index.txt, containing the inverted index for just these five bigrams, using files in devdata:

computer science

information retrieval

power politics

los angeles

bruce willis

Modify your mapper, to output (word1 word2, docID) pairs, rather than the (word, docID) pairs you had in the unigram task. There is no need to change your reducer.

Extra details

If your execution takes way too long, or crashes, you can simply make the data files smaller by deleting text, starting from the end. Each file contains 500000+ words, you can make it as small as 50000; but do keep the file count the same, ie. use all the files in devdata as well as fulldata.

You could 'test' your code using the devdata collection for unigrams as well, then do it 'for real' (in 'production mode') on the fulldata set of files. Or, for both unigrams and bigrams, you can use your own set of files, eg. a.txt, b.txt... e.txt (5 files), each with a paragraph from https://www.gutenberg.org/files/74/74-0.txt [:)] - that way your code will run quite fast and output results, so you can make rapid alterations in a shortened develop-run-debug cycle. **To develop+test your code, you can even simply use the two sample files I have in the repl.**

If you like, you can read/do the 'official' MapReduce tutorial here.

Alternative: Here is a collection of shorter data files [each file has fewer words] - you have the choice of using these instead of the 'big' files. The filenames, and the file counts are the same as for the original/previous ('big') set of files, the only diff is that there are fewer words in each. For the

bigrams, you'll get different counts compared to the originals, but that shouldn't/doesn't matter - if you use these smaller files, just make sure you're creating a bigram index that shows files and counts for 'los angeles' etc. Many files end with a chopped off word, eg. 'sabb', but that's ok. Also, if you're curious, here's how I shortened the fulldata/* files :)

Rubrics

Your (max 10) points will come from fulfilling these:

- 4 points for the unigram index entries, contained in unigram_index.txt
- 4 points for the index entries to the words mentioned in the bigrams section above, contained in selected_bigram_index.txt
- 1 point for screenshots of the output folder (the output folder is what you specify as the second argument while running the job) for the job for unigrams, and bigrams (two screenshots)
- 1 point for your code, for unigrams and bigrams (two source files)

Getting help

There is a hw3 'forum' on Piazza, for you to post questions/answers. You can also meet w/ the TAs, CPs, or me.

Hope you have fun doing the HW :)