605.744 Information Retrieval
Spring 2017 – Paul McNamee

Homework #2 (due in 2 weeks)

Implementing Inverted Files (70 points)

Inverted files are the primary data structure to support the efficient determination of which documents contain specified terms. The objective of this assignment is to process the *bible* corpus from the course web page, much as in the first assignment, but this time you must also build an inverted file that contains a postings list for each dictionary term. Your implementation should model a useable real-world indexing system; in particular, this means that your inverted file structure must be written to disk as a binary file; your dictionary must be written to disk; for each word in the lexicon you must store a file offset to the corresponding on-disk posting list; and finally, you should process the source text file only once (many real-word collections are so big that the cost of multiple scans is prohibitive).

Construction (55 points)
Process the collection and create the dictionary and inverted file. The docids are the ID fields in the <P> tags of the corpus. You may use more than one 'main' program for this assignment if you prefer, but only process the original source text one time. For example, you could write a program that reads the input file and writes out records like "apple doc1432 2" and "orange doc4 3" to represent the fact that *apple* occurs in document 1432 twice and *orange* occurs in document 4 three times. These records must be sorted (by term, then by docid) and you might use a separate program just for this. Finally, you have to write out the sorted entries as an inverted file. It may not be easiest to use three programs as in this example; however, in real-world systems, there is usually at least a scan that produces a temporary file and a merging of sorted temporary files. The reason for this is because typically records for the entire collection will not fit in the memory of a single machine. Since our electronic text is very small, it is possible to create an index using a single program without relying on auxiliary storage for this assignment. This can be achieved by following the memory-based inversion algorithm in the notes (*Algorithm A*) and directly writing out the postings after all documents (*i.e.,* paragraphs) have been read.

For full credit your inverted file should be a binary file[1]. I suggest as a baseline, 4-byte integers for document ids and 4-byte integers for the document term frequency; however, for this collection, 2-byte integers would suffice. I suggest that you store the length of the postings list (*i.e.,* also known as document frequency) with the other information in your dictionary data structure.

Note, the assignment is to build a **non**-positional index as described in Chapter 4 of IIR and in the lecture notes; you should ignore term order and term position in the documents. The results of indexing should be two files: a dictionary and a file containing the postings lists.

Testing (15 points)
Demonstrate the ability to identify which documents a word occurs in and the number of times that the word occurs in each. For full credit do this by reading back in your binary file formats. Print out the document frequency and postings list for terms: "archer", "blameless", "study", and "nuclear". Give document frequency, but do not print postings for the words "horse", lovingkindness", "Mary", and, "dance" (these postings lists are longer).

Looking ahead

- For the third assignment you will be given queries with the goal of ranking documents using a similarity metric such as the vector cosine method. To succeed on that assignment, it is crucial that you are able to reload a lexicon from disk and retrieve a postings list from the inverted file for any specified term.
- The dataset on the third assignment will be larger (e.g., up to 1 GB of text).

---

1 If you choose not to make your inverted file binary, there is a 15 point deduction. You should not use built-in object serialization packages (e.g., pickle in Python; writeObject in Java) for the postings lists. Note that the *dictionary* doesn't have to be written as a binary file, though that would be normal. If using Java, consider the class java.io.RandomAccessFile and the writeInt method in java.io.DataOutputStream for the postings file.

Submission Details
Your printout should contain:

- Your source code
- A brief description of what you did, and the format of your lexicon and your inverted file (this can in clear, easy to find comments in your code)
- Program output that indicates the number of documents, size of the vocabulary (*i.e.,* number of unique terms), and total number of terms observed in the collection (this same information was also required on the first assignment)
- The file sizes for your dictionary and the inverted file (in bytes). Is your index (dictionary + inverted file) smaller than the original text? Which takes up more space, the dictionary or the inverted file?
- Output for the test cases requested in the "Testing" section.

Questions (30 points: 10 points each)

[1] Character n-gram overlap is used for both automated spelling correction and personal name matching (i.e., deciding whether two names might be the same, a common database problem known as "record linkage"). Using a character 4-gram representation, how many n-grams do "CHEBYSHEV" and "TSCHEBYSCHEF" have in common? What is the Dice-coefficient score for these two strings using 4-grams? What is the Dice score using 3-grams instead? Which score is higher?

[2] Qualitatively explain the impact of using stemming on each of the following: (a) vocabulary size; (b) number of postings in an inverted file; (c) average posting list length? [The format of a good answer would be: With stemming XXXX {increases, decreases, doesn't change} by {a lot, a little, at all} because of YYYY].

[3] Express the numbers {17, 64, and 1000} three ways: using a 16-bit binary representation, and using the gamma and delta codes discussed in class.

For More Fun (extra credit, up to 3 meager points)

[3 points] One approach to supporting phrasal querying is to create a 'biword index' (cf. pgs. 36-38, IIR). For a query like 'johns hopkins medical institution' the query would be broken into 3 two-word phrases: 'johns hopkins', 'hopkins medical', and 'medical institution'. In addition to the regular (*i.e.,* single word) index that you created earlier, construct a biword index. (For this part, if you like you can ignore single words and only generate the two word pairs.) Report the differences in three measures between the single word and biword indexes: (a) vocabulary size, (b) time to build, and (c) file size in bytes of the inverted file.