CMSC676: Information Retrieval
Homework 4

John Seymour*
Department of Computer Science and Electrical Engineering
UMBC

April 9, 2014

The aim of this project is to create a retrieval engine for queries against a corpus of 503 HTML documents. The first assignment, the backbone of this project, compared alternative methods for downcasing, tokenizing, and calculating frequencies for all words within a corpus of 503 HTML documents. The second assignment calculated term weights for all words within the corpus. The third project created an inverted index for the terms. The first, second, and third reports are summarized below.

The original project stripped out all html tags and then threw away all non-alphabetic input, including punctuation and numbers. This had two interesting effects: first, that several contractions may have been mistaken as possessives, and second, that all hyphenated words became single tokens, instead of splitting into a token for each word in the compound. It utilized the java.util.Scanner class to tokenize the input files, which has a method for creating custom delimiters, useDelimiter(). A custom regular expression was used to separate the words into tokens and to parse out html tags. However, the Scanner class first delimits based on whitespace, which had the effect of including a few multi-word html tags into the final tokenized output. HTML entities, such as &acute for the e in "Felix", were also not decoded, leading to interesting phenomena. Finally, the program sometimes had empty tokens, due to all characters in the token being removed because of the above rules. Such tokens were deleted.

The original, optimized approach utilized the hashmap data structure, with the key being the token and the value being the frequency. The program linearly scanned each input file for tokens based on the above rules

*seymour1@umbc.edu

and incremented counters corresponding to tokens in the file to calculate frequency. Empty tokens, stopwords and all words of length 1 (i.e. a, b, c, etc.) were removed after tokenizing the file. However, words that only occur once in the entire corpus could not be removed during the original tokenization of the file: whether a word only occurs in that particular document cannot be known until scanning all documents. It is for that reason that the removal of words that only appear once in the entire corpus happened at the calculation of the tf * idf stage.

I also needed to store the token/frequency pairs, from documents in the corpus, in some data structure in order to calculate the tf * idf after scanning all documents (whereas in assignment 1 it was sufficient to print the frequencies after tokenizing the file- only local operations were required). I used an ArrayList to be able to easily iterate over the documents for calculating tf * idf. Without such a data structure, it can be easily shown that two passes through the data are required, further increasing the I/O time required to weight the terms in the document. The equation used for weighting values is still the simple variant, given by the following equation:

$$weight = tf * idf/normalizationconstant \tag{1}$$

Where TF is the term frequency in the document, IDF is the log of the total number of documents divided by the number of documents containing the term, and the normalization constant is the sum of all frequencies in the document post-removal of terms.

The third project wrote to the dictionary and postings files instead of ancillary files. After the term weights are recorded, we simply iterated over each term and found which documents contained the term along with how many, and their associated tf*idf value. The dictionary records were chosen to be in hash_order. I did not need to make any assumption based on the maximum length of a word for this corpus, as all words are shorter than the block size.

This project, phase four, could be implemented completely independently of the other projects if given the output from phase three. To calculate the relevance of a particular document for a query, the easiest method is to calculate the dot product of the terms in the query against the corresponding columns in the term-document matrix (TDM). However, memory should be conserved, especially when considering the postings file; in real-world corpera, the postings file itself will not be able to fit into memory. To calculate the dot product, then, I sum the product of each query term weight

(1.0 for unweighted queries) with the document term weight (given in the postings file). To do this, I first find the term in the dictionary, getting its location in the postings file. I then seek (i.e. linearly scan) to the location in the postings file and read the number of lines corresponding to the number of documents containing the term. I again emphasize that the entirety of the postings file is never stored in memory- only that which relates to a given query term. There may be alternative methods to perform the seek such as a binary search, but this method proved easiest. The data structure, therefore, is technically a skiplist with pointers for each column vector in the TDM to the column vector corresponding to the next query term. Furthermore, adding weights to the query was straightforward: in the dot product of the query vector vs. the TDM, instead of assigning each term uniform weight, I assigned it the weight given as an argument. Roughly, the program should run in linear time based on the number of query terms, but the overhead in starting the JVM actually dwarfs the variance based on number of inputs for reasonable queries.

1 Sample Queries

```
diet
018.html 0.03749719316705637
009.html 0.00848150797826275
263.html 0.007030723718823069
252.html 0.006029167871712871
050.html 0.0049996257556075155
152.html 0.002100579862921094
353.html 0.0012143977332512573

international affairs
219.html 0.03663835475706884
161.html 0.034326490836491926
138.html 0.026506550616344358
133.html 0.022582057904068194
125.html 0.02153657237577979
247.html 0.02050163411515879
143.html 0.019518984985456192
197.html 0.018723540456268322
205.html 0.018182068205630425
```

```
243.html 0.014788034754423024

Zimbabwe
There were no relevant documents corresponding to your query.

computer network
156.html 0.05595144956418351
060.html 0.04528122586899157
128.html 0.03008268851729521
181.html 0.027433898333382424
223.html 0.026240125593415903
380.html 0.02270963919344242
164.html 0.020813637259607005
037.html 0.020787241829160315
135.html 0.018940952029408094
064.html 0.0177426170462331

hydrotherapy
273.html 0.014333157073962533

identity theft
379.html 0.023352482742089495
380.html 0.013663812945892303
245.html 0.00827310906801338
301.html 0.006470344682554243
328.html 0.006327670093545724
298.html 0.0036401679899258875
397.html 0.0030222796964329323
027.html 0.0024869568757084756
391.html 0.0020787761363789967
292.html 0.001992212322610513
```

2 Sample Queries with Weights

```
diet 0.5
018.html 0.018748596583528183
009.html 0.004240753989131375
263.html 0.0035153618594115344
```

```
252.html 0.0030145839358564357
050.html 0.0024998128778037578
152.html 0.001050289931460547
353.html 6.071988666256287E-4

Computer 1.0 Network -1.0
060.html 0.04528122586899157
380.html 0.02270963919344242
037.html 0.020787241829160315
038.html 0.016263671736788457
159.html 0.01252418941647602
502.html 0.011337536766284798
030.html 0.008996088955856416
204.html 0.008009421134891519
124.html 0.007983667369184473
078.html 0.007932653520180098

King 1.0 Charles 0.6
308.html 0.027100351356550485
303.html 0.011461808619130653
008.html 0.010752769500350772
278.html 0.009610361349711095
304.html 0.008376381777838882
306.html 0.007254815770735
300.html 0.006180422232840227
281.html 0.005846952694222224
315.html 0.00583394940976478
025.html 0.004826325584315858
```