CMSC676: Information Retrieval
Homework 4

John Seymour[*]
Department of Computer Science and Electrical Engineering
UMBC

April 22, 2014

The aim of this project is to create a program which clusters the 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 fourth project created a search engine for queries against the corpus. The first, second, third, and fourth 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,

[*] `seymour1@umbc.edu`

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 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.

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 was to calculate the dot product of the terms in the query against the corresponding columns in the term-document matrix (TDM). To calculate the dot product, I summed the product of each query term weight (1.0 for unweighted queries) with the document term weight (given in the postings file) to save memory, by first finding the term in the dictionary, getting its location in the postings file, then seeking (i.e. linearly scan) to the location in the postings file and reading the number of lines corresponding to the number of documents containing the 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 dwarfed the variance based on number of inputs for reasonable queries.

Phase five is to execute agglomerative clustering, using the group average link method. Similar to phase four, phase five could be implemented completely independently of phases three and four, and even most of phase two. From input html files to clusters, my phase 5 clusterer ran in about 7 minutes.

The program utilizes phase 3, with most of the phase 3 project removed, as a backbone for this project. After the files are tokenized, with stopwords and singletons removed, the clustering algorithm proceeds as follows. First, all files are placed into singleton clusters. Then, while the maximum similarity between any two clusters is greater than 0.4, the program calculates the Jaccard similarity metric for each pair of clusters, finds the maximum similarity, and merges the clusters by removing the two individual clusters and adding the centroid of the two clusters to the list of clusters.

I used the Jaccard similarity metric, which is the magnitude of the intersection divided by the magnitude of the union. This simplified much of the logic; the centroid of two documents (or average) was simply the union of the two documents, and the similarity measure could be computed completely locally (i.e. there was no need for the knowledge of all the tokens). Furthermore, no knowledge is necessary about previous clusters; unlike with the cosine similarity measure where the centroid of two clusters can be different than the two clusters' averages (e.g. with three documents, in cosine similarity, the centroid is NOT the centroid of centroid(A,B) and C), the Jaccard similarity metric has the added bonus that the centroid of any number of clusters is the same as the centroid of any two subsets of clusters. However, the clusters are generally small using the Jaccard metric, as a pair of documents tends to have more words (and thus less words in common) as other documents.

Within the while loop, the similarity matrix is calculated between every pair of documents. The actual data structure in use is a two-dimensional array, where only the upper half is actually calculated. Clusters are named as they are needed, and emptied when they are merged (this is why there are empty clusters in the final list). I used ArrayLists as the major datastructure to hold clusters.

There are several interesting questions to answer with this project. The files 102.html and 130.html were the most similar, and this can be verified by introspection. Similarly, documents 001.html and 090.html are one most dissimilar pair, with absolutely no terms in common. Utilizing the power of the Jaccard index, finding the document closest to the corpus centroid is easy, as the corpus centroid is simply the union of all tokens in the corpus. Document 433.html is the document closest to the corpus centroid. Document 153.html is the document furthest from the corpus centroid. This means that document 433.html contributes the most terms to the corpus, and document 153 contributes the least, which can be seen somewhat in looking at the files.

Finally, here's a figure for the time to cluster vs. the number of documents. The time is roughly O(n^2).

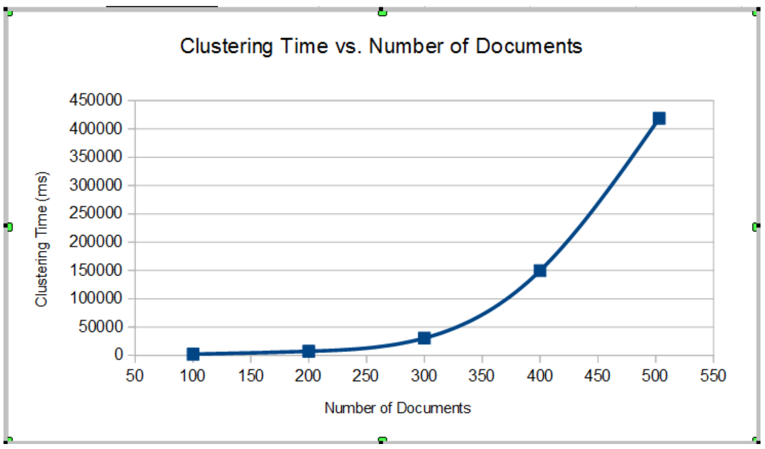

Figure 1: The amount of time required to tokenize and cluster documents for various sample sizes. All times are averaged over 10 trials on the corpus.

A First 100 lines of output

```
Merging cluster 102 and 130
Merging cluster 129 and 502
Merging cluster 418 and 420
Merging cluster 419 and 500
Merging cluster 414 and 417
Merging cluster 416 and 498
Merging cluster 413 and 414
Merging cluster 413 and 496
Merging cluster 426 and 428
Merging cluster 427 and 494
Merging cluster 400 and 402
Merging cluster 401 and 492
Merging cluster 406 and 408
Merging cluster 407 and 490
Merging cluster 421 and 487
Merging cluster 486 and 488
Merging cluster 401 and 404
Merging cluster 403 and 486
Merging cluster 419 and 484
Merging cluster 483 and 484
Merging cluster 401 and 406
Merging cluster 405 and 482
Merging cluster 476 and 478
Merging cluster 477 and 480
Merging cluster 396 and 397
Merging cluster 396 and 478
Merging cluster 411 and 412
Merging cluster 411 and 476
Merging cluster 396 and 397
Merging cluster 396 and 474
Merging cluster 396 and 467
Merging cluster 466 and 472
Merging cluster 396 and 467
Merging cluster 466 and 470
Merging cluster 65 and 81
Merging cluster 80 and 468
Merging cluster 392 and 396
Merging cluster 395 and 466
Merging cluster 143 and 174
Merging cluster 173 and 464
Merging cluster 20 and 90
Merging cluster 89 and 462
Merging cluster 71 and 458
Merging cluster 457 and 460
Merging cluster 67 and 201
Merging cluster 200 and 458
Merging cluster 3 and 73
Merging cluster 72 and 456
Merging cluster 274 and 298
Merging cluster 297 and 454
Merging cluster 445 and 446
Merging cluster 445 and 452
Merging cluster 99 and 134
Merging cluster 133 and 450
Merging cluster 233 and 243
Merging cluster 242 and 448
Merging cluster 59 and 202
Merging cluster 201 and 446
Merging cluster 5 and 21
Merging cluster 20 and 444
Merging cluster 286 and 288
Merging cluster 287 and 442
Merging cluster 365 and 366
Merging cluster 365 and 440
Merging cluster 286 and 432
Merging cluster 431 and 438
Merging cluster 273 and 280
Merging cluster 279 and 436
Merging cluster 415 and 419
Merging cluster 418 and 434
Merging cluster 265 and 283
Merging cluster 282 and 432
Merging cluster 363 and 365
Merging cluster 364 and 430
Merging cluster 257 and 278
Merging cluster 277 and 428
Merging cluster 334 and 345
Merging cluster 344 and 426
Merging cluster 99 and 106
Merging cluster 105 and 424
Merging cluster 348 and 415
Merging cluster 414 and 422
Merging cluster 258 and 260
Merging cluster 259 and 420
Merging cluster 225 and 407
Merging cluster 406 and 418
Merging cluster 148 and 166
Merging cluster 165 and 416
Merging cluster 227 and 230
Merging cluster 229 and 414
Merging cluster 107 and 150
Merging cluster 149 and 412
Merging cluster 227 and 229
Merging cluster 228 and 410
Merging cluster 372 and 380
Merging cluster 379 and 408
Cluster 0: {1, }
Cluster 1: {2, }
Cluster 2: {}
Cluster 3: {4, }
```