Patrick Trinkle
CMSC676
Spring 2009
Project 2 Write-Up

Background

An Information Retrieval system needs to determine the general content or theme of a document. This can partially be done with term weighting and stop word lists. If a term appears very frequently in a very large document (of other terms) there is a strong likelihood this term is part of the document theme. Similarly if a term appears only once in a very large document is likely non-central to the overall theme. This information can intuitively help provide better results. This idea of determining the value of a term in a document is term weighting; where a term no longer has only a raw frequency, but also has a normalized frequency correlated against the number of documents in which the term appears. This is then multiplied against the inverse of the document frequency for the term. The document frequency for the term is defined as how many documents contain this term. There is a group of term weighting methods in the category of if*idf; which stands for term frequency times inverse document frequency.

In language there are many words which provide no real content to a document and cannot aid in distinguishing a document in the corpus. These are referred to as stop words. Examples of stop words in English are {"an", "and", "ever", "has"}. Given a query for "The United State of America, " a document containing {"united", "states", "america"} is a likely match. The user query itself is likely ran through a similar parser as the documents, which will involve stripping out stop words. Therefore the query would certainly find that document. However, information such as the distance between terms in a document can provide even more accurate results.

Implementation

Due to the extremely slow nature of the homebrewed Ruby HTML tokenizer, it was necessary to rewrite it. A previous draft C# parser was ignored because it wasn't thoroughly tested. C# proved a good candidate for tokenizing because it is a systems language that has built in support for complex and efficient data structures as well as built-in string handling. C# is managed code; and therefore many details behind the data structures are encapsulated away from the user. C# 3.0 does have an interesting feature for development called LINQ. This provides for querying against data structures in a natural way in the code. Adapting this into the code allowed for trivial sorting. Because the data structures are abstracted away, complexity calculations are not entirely possible for storage and data retrieval.

Since recoding of the parser was necessary, some changes were made. Previously apostrophes as well as most punctuation was considered an end of token marker; whereby the current token would be cut. Therefore "can't" => "can" was a common break. The remaining letter "t" was dropped because of the token size (any token size 1 is not saved). Also, all information in image alternate text tags, etc is ignored; as well as link titles. Although the Information between the <a href...> is tokenized. Therefore if the HTML code is as such: Link to Mars!; it will tokenize to {"link", "to", "mars"}. There is a loss of potentially useful information because of this; because the alternate text in an image, by definition, describes the image. In this implementation all numbers are ignored and special character sequences are skipped over. Thus, "Ped&ro" => "pedro." There can be loss of a valid tag because of this, however a later implementation using n-grams would likely negate this problem.

The new parser inherits most of the details from the Ruby parser because it was modeled directly after the Ruby parser. However, the Ruby parser was implemented as a function, whereas the C# parser is an object instantiated per input file.

The stop words list must be a txt file in the input directory named "stoplist*." The stop words list is parsed in as though it was an HTML document. Therefore certain stop words will be get combined because apostrophes are ignored.

Each file is gobbled in one at a time. Stop words are pulled out as each token is written to a temporary file (this is written out but currently unused--placeholder for scaling) as well as an in-memory data structure. This process is also where document frequencies for terms are calculated. The per-input-file-stop-word-free hash list is then added onto a list of these hashes. Also while processing each file we store off the total number of terms (non-unique or instances) for each file; this is used for normalization later. Stop words are not ignored during HTML parsing because this would be less efficient. Because stop words are supposedly the most frequent the parser would have to check each token against the stop words list for each instance of the stop word, as well as every other token read in. Therefore the code saves time by only having to cross against the stop words list once per token per file.

All the information is kept in data structures stored in memory. Input files are only passed over once and the hash table created for each input file is then only walked over twice (after creation). The order in which the input files is read is maintained throughout the program for all other data processing related to the term sets. This has worked so far, but if the process was split up; it may prove useful to tag the data with the source. A dictionary could trivially key on the source file and have a value that is the index to the term list. This modification may be necessary for the next phase of development.

Once all the input files are processed; all the terms with a document count of one are removed from the hash lists. Terms that appear only once in the entire corpus can be garbage tokens. Also, since they appear only once in the entire corpus they are likely unrelated to the document theme. This involves querying against a full term list and walking through the document frequency hash list; and removing terms from each of the document hash tables in turn. This step is necessary, because it is infeasible to determine if a term appears in only one document until all are parsed into the system. Therefore, the performance hit is a necessary burden. However, there may be faster ways of walking through the terms by using specific data structures and sorting routines.

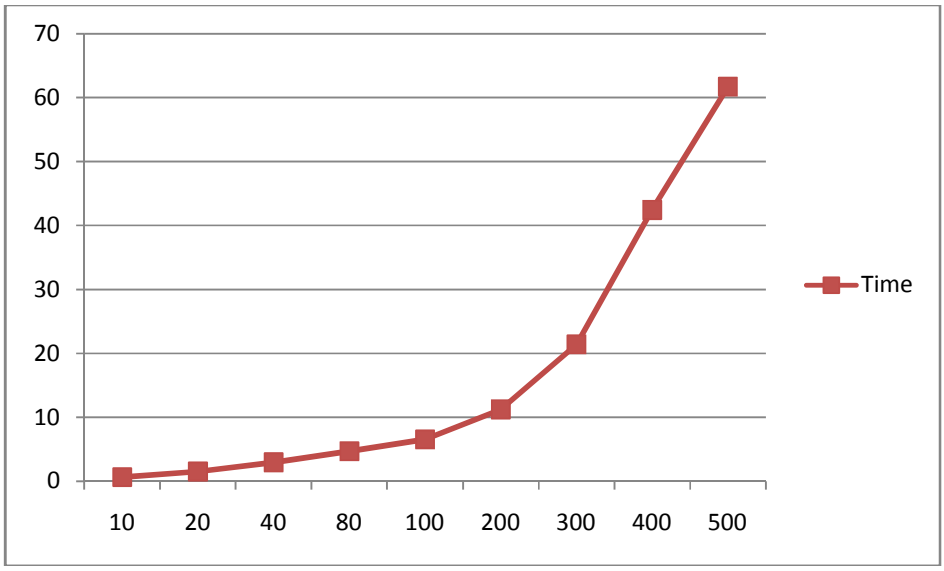
After the hash list of document counts is cleaned out, the inverted document frequency is calculated by the following equation: $idf_i = \log10(doc\_cnt / doc\_freq_i)$. The information is then used to output each file with the token, and the tf * idf token weight. The token frequencies are normalized in this step as well. They are normalized by dividing the term frequency by the total term size for the file. The output was confirmed using small sets (2-4 files) and a spreadsheet.

Conclusion

Even with the considerably larger workload of the second phase of the project, the C# implementation is considerably faster than the phase one Ruby implementation. As a proof of concept, before the extra complexity was added, timings were compared to the previous Ruby implementation and the C# was dramatically faster at the HTML parsing. The time is calculated via a built-in C# time fetcher. The program reads in the current system time when it starts and then read the time in again upon completion. The subtraction of the former from the latter provides the total processing time. This was

necessary as the program was run in the Windows operating system; and a tool such as time (for *nix) was unavailable for timing. Also of note, the breakout of system versus user time was not available.

It would follow that there should be a nearly linear growth to required time based on the input files. However, this is not the case. The steep increase from 300 to 400 indicates larger than average files in the 3xx range from the input. This detail has been confirmed. Some files in the 3xx range are considerably larger than any previously parsed. Another burden is the extra term space management required to handle more and more files. The slope between 40 and 80 is 0.04375. This is on par for that region of the graph, but the slope from 300 to 400 and 400 to 500 are 0.210 and 0.192 respectively. These are larger and indicate either greater overhead in term management or much larger input files. Likely a combination of the two.



Time in Seconds by Number of Input Files