Report

Building a Search Engine: Part 1

Jesse Geman

1. My inverted index is a dictionary. Each key in the dictionary is a token obtained from a title or text tag of the XML file (filtered and stemmed), and each value for that key is also a dictionary. These ‘sub’ dictionaries, the values corresponding to the tokens, have page ids for keys (the pages the token showed up in) and lists of locations (the location of the token in the stream obtained from that page) for values. I did not use skip pointers; I did not see a use for them. The structure of my inverted index allowed me to use json to dump it into a text file that was easily retrieved in my query program in the same format (dictionaries in dictionaries).
2. I was pretty pleased with my inverted index (dictionary); python dictionaries use hashing and worse case you’re dealing with O(n), so adding things to my current dictionaries wouldn’t be too bad, pointers for the word location lists could be useful here. Of course, I would have to write code to update my inverted index, write now my program only creates it from scratch.
3. Certain words are used often, but only a few times in a page, while others show up in only a few pages, but often when they do, while still others show up in many pages and often in that page. The distributions vary quite a bit.
4. I would examine the frequency of which words appear in a given XML corpus, and remove frequently occurring words that do little to define the page content (there might be some topic related stopwords, e.g. breaking in the breaking bad pages). Beyond this, I would probably add the most commonly used words in english: ‘a’, ‘the’, ‘and’, etc (it seems likely that many of these words would also be the most frequently occuring words in the corpus).
5. My phrase queries function first makes sure every token in the query is actually in the index (an optimization), if this is the case it starts with the first two queries and finds the instances where they occur sequentially in the same page. It then takes this set of pages and locations and does it for the next token, it follows this pattern until it gets to the last word, or finds that the tokens are not sequential. My boolean query is recursive, and takes in boolean AST expressions as arguments, tuples nested in the lists in those expressions are passed recursively to the boolean query function as arguments, while the page ids for the strings are retrieved and recorded. My function relies on basic set algebra in creating the page id list that it returns. There are a number of different scenarios I have to account for (e.g. the first element of the tuple is “AND” and one of the strings in the list does is not a dictionary key: return an empty list).