Matthew Bejtlich
2040
Part 1
Report

1. What data structures do you use for storing the inverted index? Do you use skip pointers? Explain your choice. In which format do you store the index on disk?

   I used a nested dictionary structure (dictionary in a dictionary). A python dictionary is a hashmap, with worst-case O(n) and average time complexity O(1). The dictionary provides an efficient way to read from memory and access positing lists corresponding to specific words in the corpus. The outer dictionary key represents a stemmed word, while the inner dictionary holds the document ID and a list of word positions in the document. The inverted index was stored in a JSON file format, because it provided me with a highly structured data file that I could easily read in my query.py program.

2. How would your data structures for postings change if new documents were added frequently to the collection?

   If I were adding new documents often, I would consider adding a linked list structure. Linked-lists are very efficient (O(1)) for adding new items to the end of a sequence, since the next pointer is set to this last item, with a reference typically stored to the previous last item. In contrast, adding many new items to an array can be inefficient (O(n)), since if the array is already full, it must be copied to add a new element.

3. Examine the lengths of the postings lists for the terms in the full collection and comment on their distribution

   I would compute mean and variance of the posting lists corresponding to the terms. I ran out of time to complete this analysis.

4. If you were not provided with the stop words list, how would you have created one?

   If not provided with a stop words list, I would consider using the term frequency-inverse document frequency (tf-idf) statistic to help me generate a stop-word list for a given set of documents. I would want to compute the amount of times a word appears in a given document. I would search for words with a low tf-dif score, because these are words that appear most frequently in all documents of the collection; these words would become my stop words list.

5. Please describe how you process phrase and boolean queries and any optimizations you added for faster processing.

   I used a query factory structure for generating phrase and boolean objects. However, my processing pipeline is different for each query. In the case of phrase, I first did all of my remove characters, lowercase, and stopword matching prior to matching the query to my inverted index. However, in the boolean case, my pipeline was in a different order; I first sent the un-processing query to the provided boolparser.py file and generated an abstract syntax tree (AST). Then, within the recursive function, I did my processing (e.g. lowercase, stemming, and matching) for each of the terms in my query.