Text Technologies for Data Science

Assignment 1

S1751472

Implementation

Tokenisation

To tokenise, a list of documents is looped through. For each document in the list, tokenisation is applied; the stream is split on any whitespace using the regex pattern r'[^\w+\']+'. Then this token stream is case-folded; every token is converted to lower case using Python's lower() function. This is done in the functions pre_process() and tokeniser().

Stemming

Stopping and stemming are taken care of in the same function, stopping_and_stemming(). First the text file englishST.txt is read and split into a list of stop words. The token stream from tokenisation is then iterated over, and each token that is not a stop word, and is not an empty string, is stemmed using the Python Porter2 Stemmer.

Inverted Index

I designed the inverted, and positional inverted index data structure to make the implementation of the search algorithms easier. This however, involved going back and changing the original design. The positional inverted index is a dictionary, where the keys are terms, and the values are doc_pos_lists. A doc_pos_list is a list of dictionaries, where the values are document IDs, and the values are lists of positions that the term appears at within the document associated with document ID. For a term "term", the positional inverted matrix entry will look like:

$$positional\_inverted\_index['term'] = [\{docID :' pos1',' pos2', \dots\}, \dots]$$

Note that every entry in the data structures are strings.

This data structure is crated and filled as follows. A large list, term_stream, is created, containing every unique term that occurs within the document set. This list is then iterated over, and a nested loop iterates over a list containing the token stream for each document. Every position the term occurs at within this token stream is added to a dictionary with the key being the documentID. After each document is iterated over, the corresponding dictionary is added to a list. After the term is iterated over, this list is added to a dictionary, where the value is the term. After completing the loops, a positional inverted index will be formed.

An inverted index is formed at the same time as the positional inverted index; however, instead of the innermost dictionaries (doc_pos dictionaries) having lists of positions of values, the length of the list is used instead.

Boolean Search

In the boolean_search() function, input can be a single term, two terms with a logical operator ("OR" or "AND"), one phrase with a logical operator and term, or two phrases and a logical operator. "NOT" operators can also be used, with zero or one "NOT" per term.

For a single term, boolean_search() calls the term_present_docs() function, checking if the term is present in the positional inverted index, and then returning all documents in which the term occurs. This same functionality is used in the rest of the cases, with intermediate steps.

Boolean search first checks if a phrase is present, and where this phrase occurs- at the beginning, end, or both. The query is then split into either a phrase, logical operator, and term- or into a phrase, logical operator, and phrase. Phrases are handled using phrase_search(), and terms are handled with term_present_docs() function.

If a phrase does not occur, the query is split into the first term, a logical operator, and the second term. These terms are handled above.

The two logical operators "OR" and "AND are treated separately. The documents containing the terms/phrases are retrieved- if either are empty, the other list is returned. If the operator is "OR", then the union of the two lists are returned. If the operator is "AND", then the intersection is returned.

To deal with "NOT" operators, a list "nots" is created, and if a "NOT" appears, the next term is added to the nots list. When calling term_present_docs(), first the nots list is checked; if the term is not present, the function is called. However, if the term is present, the term_not_present_docs() function is called, returning all documents not containing the term. This is then used the same as the other document lists.

Phrase Search

Phrase search takes a phrase, and calls term_present_docs() for each term, resulting in two lists. The intersection of these lists is then taken. This is then iterated over, and for each term the positions at which the terms occur in the documents are added to separate lists. These lists are then fed to the terms_adjacent() function, with proximity equal to one; ensuring the terms occur one after the other. Note that ordering is important here, the first terms list must be passed first. The terms_adjacent() function iterates over the two lists, and determines if any two positions have a difference of one, and if so the two positions are added to a list and returned after checking the remaining positions. These matches are added to a list on each iteration, finally returning the list.

Proximity Search

Proximity search is a special case of phrase search, where the query is parsed, the proximity, first term, and second term are extracted, and passed into the phrase_search() function. This proximity will also be passed to terms_adjacent(), checking the proximity is less than or equal to the passed in proximity, and returning the matches.

Ranked IR

Ranked IR is implemented in ranked_retrieval(). This function iterates over each document, calculating the total number of terms in the document. retrieval_score() is called on each iteration, which, for each term in the query term list, calculates the df and tf values and increments the score by adding the weight, calculated from the term_weight function. The weight is calculated by first calculating the left hand side of the equation, $1 + log(tf)$, followed by the right hand side, $log(\frac{N}{df})$, and multiply both sides together. Each score is recorded and returned.

Evaluation

I am happy with the overall implementation and design of the system. I am not, however, happy with the time it takes to load. Currently to parse the provided document list the system takes approximately forty minutes, and an additional forty minutes per ranked IR query. I ran out of time to query these, and therefore do not have a corresponding output file submitted.

I attempted to fix the processing time of the system by changing the positional inverted index structure; it was originally the same as described above, however the doc_pos dictionaries were lists, containing just the positions and not the document numbers. If the term did not appear in a document, an empty list was added, and the document numbers were deduced. This worked well on test data, but after scaling I realised it resulted in parsing a huge amount of trivial data, mostly empty lists. To combat this I implemented the data structure described above, and reduced the run time by around twenty minutes. As a result, I was required to write almost the entire system, from the pre-processing, to the search functions, taking a huge amount of time. If I were to try to further optimise the system, instead of a large amount nested loops being used for matches, I would utilise regex pattern matching, hopefully resulting in quicker searches.

I also faced issues implementing the search functions. I misunderstood the task, and implemented the functions to return True or False values for the queries. This misunderstanding resulted in me also having to rewrite the whole search system.

If I was to do this task again, I would focus on time management, properly understanding the task description, and spending more time planning what data structures to use.