Text Technologies for Data Science Coursework 1 Report

Vasileios Ntogramatzis

October 2020

Tokenisation and stemming are both handled in the preprocess() function. The function accepts a string of tokens as input. The string is split on all non alphabetic characters using a regular expression and then all spaces are removed. Tokenisation is done in this way not only because it is quite simple and efficient to implement but also because it allows for matching tokens in other documents or in queries easily. Splitting on non alphabetic characters also removes all punctuation which is something that can vary across documents and if it is not removed may result in false negatives. The resulting tokens are then converted to lower case. Next, all stop words are removed from the list. The stopwords are obtained from the stopwords.txt file that should be saved in the same directory as code.py. Since the stopwords are already lower case and contain no punctuation characters except the apostrophe, only the apostrophe character is removed from the stopwords. This is done to ensure that the stopwords correctly match any stopwords in the tokens list. After stopwords are removed the resulting tokens undergo stemming. Stemming is done using Python's PorterStemmer. PorterStemmer is imported from the nltk library. The stemmed tokens are then returned. In summary, the preprocess() function removes punctuation from the input string, splits the result on space characters, converts the resulting list of tokens to lowercase, removes stopwords and then performs Porter stemming.

The inverted index is created in the index() function. The index itself is a dictionary where every key is a term and the value corresponding to each key is a tuple, where the first element is the document in which the term occured and the second element is a list of all positions in that document in which the term appeared. The function accepts as input a dictionary where each key is a document and the corresponding value is a list of all tokens appearing in that document. For every document, the function iterates over all the tokens in that document and adds them to a local index where the key is the term and the value is a list of all positions of occurence. This local index is then added to the global index of the form described above. Once the loop iterates over all the documents in the input, the global index is returned.

Code.py includes search functions for boolean search, phrase search, proximity search and ranked IR based on TFIDF. Boolean search, phrase search and proximity search are all handled in the queries_search() function which in turn uses various helper functions. The queries_search() function accept a list of queries and an index. The logic of code.py reads in a file containing the queries and splits them by line, creating a list of queries to be fed as input. For each query in the list, the function performs search using a regular expression to extract the number of the query and the query itself. The query along with the index are then fed into the auxiliary function search(). Search is a recursive function that accepts a query and performs boolean search, phrase search or proximity search depending on the query. The format of the query is determined using regular expressions.

If the query has the form of a proximity search (e.g. #10(income, taxes)) the words in the query are preprocessed using the preprocess() function and then passed into the proximity_search() function along with the index and n the proximity distance for the search. The words in the query and n are both extracted from the original proximity search query using regular expressions. The proximity_search() function works by first checking that both words in the query exist in the index. If they do, the function iterates over all tuples for the corresponding terms in the index for both words. This is done in a nested loop where the first loop iterates over all tuples corresponding to the index entry of the first word, inside of which a second nested loop does the same for the second word. For the two tuples in the current iteration of the nested loop, their first values, equal to the number of a document the words occured in, are compared. If they are equal, the two words have appeared in the same document. This leads to a second nested loop where each loop iterates over the second element of each tuple, the list of positions the word occurred in for a document in which both words occured. The outer loop iterates over all positions for the first tuple and the inner loop for the second tuple. For each pair of positions in the current iteration of the nested loop, their distance is calculated. If the distance is less than or equal to n and greater than 0 (as we do not want the second word to appear before the first) then the document number is added to the list of results. If, the position of the first tuple ever exceeds the position of the second and since the two lists of positions are ordered, then we skip the current iteration. This is done for efficiency as it is impossible to find a match if the first word does not occur before the second. Once all loops complete, the result, a list of all documents satisfying the query, is returned.

If the query does not have the form of a proximity search, then search() tries to determine if the query is a boolean query using regular expressions. If the query is of the form [A OR B] or [A AND B], then search() extracts the subqueries A and B and recursively runs search() on each of them. In the case of an AND query, the common results of the two recursive calls are returned as one list (i.e the resulting list is the conjuction of the two lists returned). And in the case of an OR query, the results of the two recursive calls are merged into one list (i.e. the resulting list is the disjunction of the two lists returned). If the query is of the form [NOT A], search() again recursively calls itself on A and negates the result by subtracting the results of the query from the list of all unique documents. Finally, if the query contains no logical operators, the function runs preprocess() on the query and determines the number of tokens in the query. If the number of tokens is equal to 1, search() calls the word_search() function and if the number of tokens is equal to 2, search() calls the phrase_search() function.

The word_search() function accepts as input a token and an index. The function checks if the token exists in the index and returns the list of documents corresponding to that term in the index. The function does this by extracting the first item in each tuple (i.e. the document number) in the list of tuples corresponding to the term key in the index. The phrase_search() function accepts as input a list of tokens and the index. The function checks if the two tokens exist in the index and if they do obtains the list of tuples corresponding to the token key in the index. A nested loop iterates over all the elements in the list of tuples corresponding to the first word and the list of tuples corresponding to the second word. Each iteration compares the two tuples and if the first element of the two tuples, the document number, are equal then the second elements of both tuples, the list of positions, are compared. The comparison is done by adding one to each position in the list from the first word and comparing the resulting list to the list of positions of the second word for the current document. If the two lists have any elements in common it means that the phrase appears in the document since the position of the second word will be equal to one plus the position of the first word.

Ranked IR based on TFIDF is handled in the tfidf() function. The function accepts a list of queries, the index and a dictionary where the key of each entry is a document number and the corresponding entry is the list of all tokens in that document. Again, the logic of code.py reads in a file containing the queries and splits them by line, creating a list of queries to be fed as input. For each query in the input list of queries, the function extracts the query using regular expressions and creates a list of tokens in the query by calling preprocess() on it. Then, for each word in the extracted list of tokens, the function checks if the token exists in the index. If it does, the function iterates over all the tuples in the token entry in the index. Each tuple corresponds to a document, so for each document the query term appeared in, the weight of the term in the document is calculated using the $w_{t.d}$ score using the following formula:

$$w_{t.d} = (1 + log_{10}(tf(t,d))(log_{10}\frac{N}{df(t)})$$

Here, N represents the number of documents in the collection and is calculated by taking the length of the input dictionary (of all documents and their tokens). $tf(t,d)$ is the term frequency of t in d, which here is calculated by taking the length of the second element of the current tuple (i.e. the list of all positions the term appeared in the current document). Finally, $df(t)$ is the number of documents this term appeared in and is calculated by taking the length of the list of tuples corresponding to the entry of the query term in the index. The weights of all term weights, for each term in the query, for a document are summed. The documents are then ranked in descending order of cumulative weight and added to a list of results. This is done for all queries and the resulting list of ranked documents for each query is returned.

Overall, this project was a very important lesson in understanding the scale of decision making involved in IR. Almost every part of the final system involved careful consideration of not only the method to use for a particular task but also how to most efficiently implement it. For example, when removing punctuation in tokenisation I considered first preserving the apostrophe character for words like "don't" but then realized there were instances of words wrapped in the apostrophe character in the text that remained intact under tokenisation. Here I had to make a decision as to which outcome would lead to better results, a consideration that is not always immediately clear. Through this project, I also realized that implementing such systems requires very complex data structures. The index alone required very careful consideration both in its design and implementation and in its use in other functions. Using Python was very helpful as it is a language that does not impose strict oversight on types, but I can imagine that this process would be considerably more difficult on a stricter language like Java or even C. Overall, I believe the system I implemented is fairly efficient and I tried to optimise the running time where possible. The algorithm runs within a few seconds for large text collections. However, because of the scale of the final code, it is unclear how such a system scales to more massive texts. On a technical level, I learned how to use regular expressions, something I previously did not know how to do. Initially, while implementing many of the query decomposition components I used more inefficient search methods on the query strings. Integrating regular expressions helped to reduce the code and make it run more efficiently.

A key area of challenge I faced was implementing the search functions. Specifically, query decomposition was particularly difficult. I initially attempted to create a function that can work for long queries with multiple operators by using a stack that stores operators. However, I found this task very difficult and reverted to creating a search function that works only for queries with at most one AND or OR operator as per the specifications of the coursework. I also faced a challenge when implementing proximity search and to a lesser extent phrase search, because of the many nested loops that made keeping track of all the variables confusing. If I were to re-implement the system, I would try to implement operator stack with a more premeditated approach as I now have a clearer view of all the components of such a system. In addition, in dealing with loops I would try to rewrite the code to be more functional both for simplicity and efficiency.

On a more theoretical level there were some difficulties with edge cases in the text. For example, words like the month "May" were removed in preprocessing because the word would be converted to lower case and then removed as "may" is a stopword. This and other such cases means that my system is imperfect as it may often remove key words, thereby reducing the precision of the search algorithms. On any future re-implementations, I would create a more involved preprocessing function, while ensuring that it remains simple and efficient in order to handle such cases. However, as a first implementation, the system performs fairly well and the learning experience of implementing it has given me the knowledge and foresight to create better systems in the future.