Text Technologies for Data Science Coursework 1 Report

1. Tokenisation

For tokenisation, I simply used the nltk tokeniser. I searched for many ways to tokenise text but I felt that the module provided by nltk was the quickest and most efficient.

2. Stopwords and Stemming

I used the stopwords given to me in the list provided on the coursework page. I also used the porter2 stemmer in python for stemming the words. Here is the first installation instruction:

```
pip install stemming==1.0
```

3. Inverted Index

For the inverted index, I first extracted the ID numbers from the text file containing the documents and stored them in a dictionary with the IDs as the keys and the HEADLINE and TEXT lines combined together as values.

I then applied all the preprocessing to each of the individual values in this dictionary (which are basically the required text for the individual documents). I created another dictionary called `dict` which has the word as a key and another dictionary as a value, which in turn has the document ID as a key and the list of occurrences as a value. It is represented like this example:

```
{apple: {5015 : [4, 67, 89]}}
```

This was achieved by parsing through each of the preprocessed documents one at a time and checking if the word is already not there in the dictionary. I then wrote all of the items in the dictionary to the preprocess text file as required.

3. Search Module

I felt that the search module was the hardest bit of the assignment to implement. My search query deals with phrase search queries, individual word queries, proximity search queries and queries containing AND, OR and NOT separately.
The module splits the AND, OR and NOT containing queries on AND/OR and then has separate conditionals to deal with each of the cases. If the term has a NOT, the module will extract the word out of this term and find all the documents that contain the word, and then subtract the list of documents containing this word from the list of all documents (`idlist`).

Phrase search and individual word search are dealt with the same way inside the conditional for queries containing AND/OR/NOT as they are for queries that are simply a phrase or a word. For phrase search, the documents containing both the words are found, with their occurence locations. The module then checks if the any of the locations of the first word + 1 are equal to the locations of the second word. If this is true, we have found the phrase and we add the document ID to our list. For individual word search, the list of documents (keys) containing the word are simply added to our list.

If the query contains an AND, the lists of documents are converted to sets and Intersected with each other. If the query contains an OR, the lists of documents are converted to sets and a union is taken. The document IDs are then written to a file in the required format.

For proximity search, the locations in a document containing both the words are computed similarly to how we did in phrase search. The module computes an absolute value difference between all the locations of both words and checks if this is less than the number specified in the proximity search. If yes, the document ID is added to our list of documents.

3. Ranked Retrieval.

The ranked retrieval module took the least time to build. It extracts each of the queries and preprocess the words accordingly, removing any punctuation. It then loops through all the document IDs and below that loops through the list of words.

If it finds the term in this document, it calculates the document frequency as the length of the list of keys for this word (all its documents IDs) and the term frequency as the length of the list of occurrences for this document ID. It then adds the computed term frequency document inverse frequency to the summed score for this document. It writes the documents and scores in inverse order of the scores for each query.

4. System comments and Learnings

I feel that the system we built here is inefficient in terms of scaling. In general:

1. I used too many loops and conditionals, which obviously increased the run time. I feel that my code was quite efficient but it still took long to run. The system could have been made more efficient by also thinking about space and time complexity.

2. The system could be scaled by increasing the complexity of queries searched for, possibly by adding AND with a proximity search or similar ways. We could also use different scoring systems like tfidf to see how the documents rank with different scores.

3. I feel that this was great challenge to build a complex text processing system and gave us great insight into how something as complex as Google works. We had to use our extensive knowledge of python to build this system. It took me over 3 full days to build, debug and comment the entire system, including thinking about how everything is going to work (data structures and code blocks).