Text Technologies For Data Science Assignment 1: Report

Introduction

This report will discuss the implementation of a basic IR system. The system is capable of parsing a given XML file containing documents of text then performing basic pre-processing functions such as: tokenisation, stop word removal and stemming. As well as this it can also take several search queries such as: Boolean search, proximity search, phrase search and ranked IR retrieval based on Text Frequency Inverse Document Frequency (TFIDF).

Tokenisation & Stemming Methods

Tokenisation

For tokenisation, it was decided that tokens would be parsed by splitting the text on each non letter character (except from ‘ and -) which can preserve useful context when it comes to queries. Furthermore, tokens were made lower case. This is standard practice for word tokenisation. The system has its own tokenisation function which takes a dictionary with keys equal to the doc number and value is the headline and body of text, for each document in the XML file. The function iterates through this dictionary and for each body of text it splits the words into tokens using a regex expression which can be found at line 63. This returns a list of words. Finally this list is iterated through and turned to lower case using the built in ‘lower’ Python function. This was done for every document and stored as a dictionary with the doc number as the key and the tokenised text as the value.

Stemming

To stem the words, it was decided that a porter stemmer would be a good choice. This is a well-known stemmer for English text and has a built-in function to the nltk Python toolkit. For this reason, Porter stemmer was chosen. The system has a function for porter stemming named ‘Pstem’ this can take several inputs: A dictionary with arbitrary key and list of string, a list of strings and a string. This allows for the function to be used in many different situations in which it may be required. The stemmer works by iterating though each string instance for each input and calling the porter stemmer function from nltk.

Building the Positional Inverted Index

A positional inverted index is a data structure for representing positions and document frequencies of terms one or more bodies of text. It was decided that the most concise way to represent this in the code was by creating a data structure of the following format: (Dictionary[String -> term][List -> Doc Frequency, Dictionary[Doc ID][List -> Positions]]). Building the index was a somewhat tedious process and assumed that all pre-processing on the text was complete. The system has a function which iterates through a dictionary which should consist of keys which represent the document ID and value which represents pre-processed words. At each words it decides if the word is already in the index, if so it updates positions and document frequency (if unseen in this document before), otherwise it will create a new entry for this word and initialise positions and document frequency.

Search Functions

Boolean Search

One of the searches this IR system can handle is Boolean search. This allows queries in the that have one ‘AND’ or ‘OR’ operator in them as well as a ‘NOT’. The queries should have two search terms. This function was implemented by first parsing the string query and searching to determine the Boolean operators within the query (if any) this was done by calling back a contains check on a string in Python for ‘AND’ ‘OR’ and ‘NOT’. Once it was determined then the terms had to be parsed for querying so the system would split the query string on these operators. This would give the system two strings containing the exact terms to query. These would be passed through the pre-processing function, then queried against the positional inverted index. What is returned is the document IDs which satisfy this query.

Phrase Search

The IR system can also query phrases (multiple terms in some order) these could be considered a term as a whole and therefor used in Boolean and proximity searched. Searching for a phrase in an inverted index requires that the order of the phrase is preserved when pre-processing the query. This is because the search looks through each document in the index for each term in the phrase and checks if the positions of the words increment from one at each term. The phrase search was relatively simple to implement as the queries could be parsed by a space and the index search was straight forward.

Proximity Search

Proximity search asks if two terms are within some number of index positions from each other. This was perhaps the most difficult to implement. Parsing the queries require a little more complicated regex that just splitting on non-letters as phrases could be included so the space in-between these terms had to be accounted for among other issues such as parsing the number in the query representing the proximity. Once the terms were parsed the querying was not too bad. For phrases the index position of the second term was used and for single terms whichever position it was discovered on finding the difference between these two positions and checking this was less than or equal to the proximity was all that was required to return a result.

Ranked Retrieval TDIDF

This method of ranked retrieval uses a formula including term frequency and document frequency. This was a simple function which is implemented by the IR system. This iterates through one or more queries and assigns them a score for each document based on a scoring formula. For each query the 150 top document scores are returned as search results.

This System as a Whole

Implementing this system was at times very difficult and I found that it is very easy to make mistakes and test your results when dealing with such large amounts and creating large data structures. This taught me the importance of testing regularly as develop each stage of the system because more often than not small errors and bugs would built up and be responsible for issues in results. If these bugs are not discovered early on then they become very hard to find once you have written many more lines of code. Another challenge I faced when implementing this system was parsing queries. I found this a challenge as it was sometimes difficult to create a correct regex expression to return exactly what I was looking for. However, this was down to lack of experience.

Improving & Scaling the System

To improve this system, I would suggest making this code more efficient by finding areas which could take advantage of built-in optimised functions. The search functions use a lot of nested for loops which have a very high time complexity when running for large amounts of data. The Boolean search checks for the 'NOT' operator multiple times in the query. This is an inefficient area of the system which should be fixed if the system is to be scaled.
Finally, for improving the system, I suggest that the query parser be more flexible. Currently, queries must follow a very strict structure and if this is not followed, the system will fail to parse the query. This makes the system not so user friendly and difficult to use. The code currently requires that queries are passed to a file in a 'Collections' folder as a .txt file. This is also quite a restrictive property of the code, and a useful improvement could be adding the ability to write queries as arguments from the command line.