Robertas Dereskevicius

TTDS Coursework 1 Report

Preprocessing

Before words within articles can be indexed and queries can be run, preprocessing must be run on the text data which splits sentences into words, modifies and filters them. The steps listed below have been applied to all the 5000 coursework headlines and bodies. Additionally, the identical procedures were used in modifying raw queries. As the inverted index stores processed data by default, the queries must be modified accordingly to match and produce good results. This has a singular exception, where for boolean queries a different form of tokenization has been applied as this type of query has inherently different parsing rules (AND, OR, NOT, “”, #X()).

Tokenization

Tokenization used within this assignment employs the basic principle of splitting terms on non-alphanumerical characters. It utilizes a “for” loop which iterates through each character and adds it to the current word. If a character is non-alphanumerical then the word is added to an array and gets set back to an empty string. Additional edge case handling measures have been implemented for **-** and **‘** symbols within words:

- If the word so far is no longer than two **letters or numbers, but not numbers only** and the current character is **‘** or **-**, then the non-alphanumerical symbol gets ignored and characters after it get added to the same word. This is important for words which have a short prefix to them which would not make any sense if it was made into a separate token and would change the meaning of a word if it was removed (O’Neil, co-ordinates, co-operative, re-defined).
- If the word so far is no longer than two **numbers only** and the current character is **‘** or **-**, then the prefix number gets split as a separate token. The numerical sequence holds meaning and can improve the accuracy of results when looking for a date or age (34-year-old, 31-september).
- If there is only a single letter after **‘** or **-** before reaching another non-alphanumerical character, then it is scrapped. The letter holds no meaning by itself and does not significantly influence the word that it would get attached to after it has been stemmed (Robert’s, aren’t, don’t).

The previously made choices have all been made after carefully observing the input set of articles and deciding which choices would impact the query accuracy in the most positive way.

Lowercase

This was the easiest step of preprocessing only involving the usage of a single python built-in function “lower()”. All capitalized letters have been substituted for the lowercase counterparts as the process of querying could result in incorrect results for frequent cases. An example case could be where words start at the beginning of a sentence with a capitalized letter. The basic implementation produces false results in the cases where a name of a city or person could also refer to an object as well. However, the positives outweigh the negatives by a large margin.

Stopping

Stop word removal has been applied to the documents and queries. The provided text file containing them has been used. While natural language processing may require the stop words to convey intention through text, indexing and querying do not need them as much. This is especially true in the case of Ranked IR where the stop words barely impact the score as they are used so often that their Document Frequency is very high. The excluded words have been stored inside a set structure with an access time of O(1). As such, any set of tokens can be easily filtered with the use of a single-line “for” loop.

Stemming

Porter Stemmer is used for stemming of all tokens (for both FT Articles and queries). It contained within the NLTK library which is imported in both assignment scripts. The use of it is quite simple as all one needs to do is initialize a “PorterStemmer” object and call “stem(word)” function. It can only be run on individual words. As such, a one-line “for” loop has been written to handle the stemming process. It introduces an error to the overall process of Information Retrieval. As an example, words such as “universal”, “universe”, “university” have all resulted in the same stem “univers”. This, unfortunately, results in loss of query precision. However, it seems that the positives outweigh the cost as the results are typically more accurate overall after stemming application which was stated during one of the TTDS lectures.

Inverted Index

The primary data structure which houses the inverted index module is quite simple and intuitive. It utilizes the python dictionary which can be otherwise be thought of as a map. This is incredibly useful during query time as the access time is O(1). The index can be described in further detail as a dictionary of dictionaries of integers where:

1. The first dictionary uses the term as a key and stores a dictionary of articles as dictionaries.
2. Second nested dictionaries use article ids as a key and contain an array of integer values.
3. The array of integer values represents word positions inside an article after preprocessing has been applied.

Access of each term position can be done using the following structure:

“InvertedMap[term][articleID][positionID]”

A dictionary function InvertedMap.keys() can always be called to retrieve a set of map keys. Iterators can be used to traverse through this data structure as well. The map is only sorted for the newer versions of Python 3. As such, sorted() function is used on it before it gets saved to make sure that it results in the same index file no matter what software the user utilizes.

Boolean Search

The boolean search utilizes a slightly different approach to the preprocessing. It reads the query string from a file and then passes it to a function which carefully parses the query one letter at a time. This is done as Boolean search has a different query structure compared to a simple Ranked IR search. The code utilizes a “for” loop for reading the query string and stores results of query elements in a stack. If a word is read, then a list of article ids that contain it gets added to the stack. Before article ids get retrieved for any of the words, phrases or proximity queries, they get preprocessed using lower-casing, stopping and stemming. If another word is read afterwards and the stack is not empty, it is assumed that a logical operator has been read in between them. Arrays of ids for both of the words then get linearly merged and filtered depending on the operator. The result is a brand new array of article ids that satisfies the criteria and replaces the current elements of the stack. This holds true for cases of phrase or proximity search as they both simply return a list of article ids just like a single word would.

In the case of a proximity query, the absolute distance between two relevant terms is computed. This can be achieved by simple dictionary iteration to find article ids which contain the relevant terms and retrieving their positions. Phrase implementation is quite simple as it is simply a proximity query where the distance from a second word to the right must be exactly 1 in the same document.

Ranked IR

The Ranked IR implementation is simple and short. The queries are preprocessed using all four of the previously described steps. The resulting terms are iterated through and a map of weights is retrieved for each of them. The map uses article ids as keys and houses the TFIDF value which was calculated using:

*(1 + math.log(termFreq),10)) * math.log(numDocs/docFreq,10)*

The resulting maps for each term are retrieved and summed to a singular global map which defines the final score of each article in relation to the current query. They get sorted in descending order and printed out to a file.

Interface & Output

The user is presented with an ability to select the names of input files which are used for index creation, index reading, querying and stop word removal. The exception being names of the output files which are all defined already. The software is built as a command-line tool. It prints out feedback to the user before and after any major processing has been done. An additional option is given to modify preprocessing that is applied to the articles or queries. Stop words and stemming can be disabled, however, they are enabled by default and should only be turned off for experimentation purposes. Additionally, one must make sure that both scripts use the same preprocessing as different settings will result in poor query results. The query main menu contains a simple numerical set of options where a user can either use Boolean search, Ranked IR or Exit.

Development & Challenges

The development was primarily based on weekly lab exercises. It all began with preprocessing implementation, then evolved to handle inverted indexing and, finally, began supporting querying. A minimum viable product was written which contained all of the most basic required functionality. I then proceeded to improve the performance of existing code and completely rewrite Boolean search logic as it was imperfect. Throughout the course of development, I was frequently looking at questions and suggestions on Piazza and implemented some of the better ideas. Posted lab results of other individuals were incredibly useful. I had fixed errors in my code after seeing a heavy mismatch between query outputs of other individuals and me.

The main surprising challenge I had to overcome was the implementation of Boolean query parsing. Ranked IR was easier to implement for me. The reason for that is that I had initially struggled to come up with an overall query handling mechanism which would combine logical operators, phrases and proximity search. I had decided to have the results and operations stored in a binary representation in relation to all the articles. This was a poor decision as it scales terribly with many documents. It took me a while to think up of another better way of handling the queries. I had browsed through lecture notes and remembered about linear merge which is significantly faster. It was programmed soon after and ended up being a much more robust implementation.

A slightly smaller, yet annoying issue was the performance of indexing. Before any changes have been made my machine would take on average 70 seconds to build an inverted index map and save it for 5000 documents. It was painfully long as I began to imagine just how little this sample was compared to all the possible sets of documents. I had improved the run time to 27 seconds prior to submission by restructuring preprocessing. It was a process of trial and error and the most effective solution ended up being one-line "for" loops for each of the preprocessing steps excluding tokenization.

Improvements

The current implementation of indexing is quite basic. I believe it can be improved with techniques such as delta encoding. As the maximum number of documents for our assignment was only up to 5000, it was not necessary for this case. If we were to have 100 million articles then delta encoding would be a necessary and easy to implement step. However, if we had that quantity of data, another improvement would need to be made. Currently, it takes on average 27 seconds (personal machine) to index 5000 articles. That means 100 million articles would take 150 hours to index on average which is an incredibly long time. I am sure that further improvements could be made in preprocessing by utilizing performant NumPy or Pandas library functions.

A potential change could be made to the TFIDF formula which, at the moment, does not contain any form of normalization. Additionally, other variants of term frequency and document frequency may be more suited for this set of data. It is difficult to judge as relevance is subjective and we would need to have people judge which results fit a query the best.

A final set of edge cases that concern me are related to tokenization. Internet links, multi-word landmark names, names of individuals mostly get split and processed as standard terms. This results in an obvious loss of information which could be prevented with the implementation of more advanced handling of edge cases.

Conclusion

I believe that the system functions quite well and takes some of the tokenization edge cases into account. I have personally run the software on some queries and then read through the documents which were deemed the most relevant. The articles did, in fact, correspond to the query well. It surprises me how effective simple implementations of algorithms can be. I have improved the performance of the indexing to a good degree and made sure querying does not take longer than a few seconds. As such, I believe that the two sets of scripts have been completed well and could perform the task of Information Retrieval in a basic search engine system. I have learned quite a lot regarding the necessity of preprocessing in textual data. Additionally, I have covered the field of inverted index creation and learned a lot about the unique aspects of handling different types of queries. This introductory knowledge helps me understand slightly better how giant search engines such as Google or Bing function on a low level. I believe that these basics of text processing will aid me significantly in the future when I begin my job as a Data Scientist.