UNIVERSITY OF EDINBURGH

TTDS Coursework 1

Ragnor Comerford

Student Number: s1614102

October 24, 2020

1 Tokenisation

Tokenisation was performed using a regex search with the query `[\d]+[\.]?[\d]+|[\w][\w\'\-]*`. It accepts any decimal number or word starting with a letter, followed by other letters, hyphens or apostrophes. This allows us to recognize contractions and possessions such as *don't*, *won't* or *Mary's*. Hyphenated words are treated as single tokens in order to correctly identify words such as *co-education*, *build-up* or *anti-racist*. However, this is not unambiguous because there are cases where hyphenated words should be treated separately. For example, flight itineraries are often advertised like *Edinburgh-London* and should be treated separately. Nevertheless, in most cases treating hyphenated words as a single token will give reasonable results. Decimal numbers were included in the tokenisation because they can useful in search queries when the user wants to retrieve documents that contain specific numbers such as years or temperatures.

After tokenisation we also strip quotes from the left and right end of the tokens in order to properly tokenise words that are part of direct speech in the original text. (E.g. *"[...] raising ventures at present", said one broker*).

2 Stemming

Stemming is performed using the de facto standard porter stemmer in the NLTK package. We transform each token using the `stem` method in the `nltk.stem.PorterStemmer` class.

3 Inverted Index

The inverted index is implemented as a class in Python and accepts a collection XML file in the constructor from which the index is constructed. Internally, we build the inverted index using a nested dictionary. The first-level keys are the pre-processed (tokenised, stop-words removed, stemmed) terms found in the collection and the second-level keys are the documents numbers mapping to the positions where the specific term was found in the documents. The dictionary is a useful data structure as it allows us to retrieve matches for a term in $O(1)$ and build the index in $O(n)$ or $O(n \cdot m)$ with stop-word removal. First, we parse the xml file using the `xmltodict` package. Then, we iterate through the documents, retrieve the pre-processed terms and add the positions to the dictionary.

4 Search Functions

For the search queries we define the following grammar that describes the syntactic elements of the query:

```
query: PHRASE | TERM | boolean | negation | proximity
proximity: "#"NUM"("TERM","TERM")"
boolean: query OPERATOR query
NUM: /[\d]+/
PHRASE: /\"[\s\w]+\"/
TERM: /[\w][\w\-\']*/
OPERATOR: "OR" | "AND"
negation: "NOT" query
```

Query strings are parsed using an implementation of the Earley algorithm which can construct an abstract syntax tree(AST) in $O(n^2)$ for our unambiguous grammar. The AST is very useful and efficient for evaluating the queries as it allows us to define operations on the index for each rule in the grammar. These operations return search results from the index which are then passed up and transformed in a bottom-up fashion from leaves to root.

4.1 Phrase and Term Search

The search using a single term is fairly simple. We apply stemming to the term, pass it as a key to our inverted index dictionary and return the set of keys of the nested dictionary containing the document numbers.
For the phrase search, we perform a term search for each word in the phrase and then retrieve the intersection of the document numbers. Then, we iterate through the documents and their positions and evaluate if each term of the phrase search is located at positions next to each in order. Any arbitrary phrase length n is possible in the implementation, subject to memory and time constraints.

4.2 Boolean Search

The boolean search consists of a boolean operator and two queries. The grammar implementation allows any kind of composition of boolean queries. First, we evaluate the left and right queries and then perform the union of the results for the OR operator and the intersection of the results for the AND operator.
The query itself can be a term or phrase whose results are retrieved like in the section above or it can be a negation of a query in which case we return the complement of the results of the query.

4.3 Proximity Search

In proximity search, we retrieve the results of the respective terms as described in section 4.1 and perform the intersection of the results. For each document in the intersection, we build the cartesian product of the positions in the document for the first term and the second term and filter out all items where the absolute difference is bigger than the specified number in the query.

4.4 Ranked Search

For the ranked search we perform a term search for each term in the query and then iteratively build two dictionaries. One dictionary for the document frequencies where each term in the query maps to the number of documents it occurred in and one nested dictionary for the term frequencies where each document maps to a dictionary containing the frequencies of each term in that specific document.
Using the term and document frequencies we build a scores dictionary for each document that satisfies the query using the formula

$$\text{Score}(q, d) = \sum_{t \in q \cap d} (1 + \log_{10} tf(t, d)) \times \log_{10} \left(\frac{N}{df(t)} \right)$$

where $tf(d, f)$ and $df(t)$ respectively correspond to the term frequency dictionary and document frequency dictionary as described above.

5 Comments

Overall, implementing this query engine was a very rewarding learning experience as the challenges of the implementation forced me to critically think about the system, both from an engineering and user perspective.
A number of engineering challenges were related to space and time complexity that required to evaluate different data structures and restructure the code and algorithms. It was also critical to properly structure the code with pure functions and stateful and abstract classes in order to allow future developers to add easily add functionality or improve the algorithms. Although the use of a context-free grammar in my query parsing implementation was not required by the instructions, it was very useful as it makes it very easy to add new types of queries or change various parts of the query algorithms.

The user challenges were more related to the pre-processing of the text as different implementations of tokenization, stop-word removal and stemming yield different

results for the queries the user entered. E.g., whether to treat certain words as the same or different tokens has an influence on the results of the queries and depends on the intent of the user.

Ideas on how to improve and scale the implementation in the future:

- Parallel query execution (e.g retrieve results for subqueries in parallel)
- Distributed indexing process
- Distributed index storage