1 Introduction

In coursework 1, we are asked to implement basic search engine modules including **preprocessing, indexing, boolean search and ranking**. In this report, I will explain my implementation of each module and emphasize important algorithms with code and details. Moreover, I will comment on the overall quality of my system, discuss some challenging issues I met and what can be improved further.

2 Implementation

2.1 Environment

Code has been examined under DICE condition using Python 3, all the submitted output files were generated on DICE. But the reported execution time was generated on a Windows laptop with Ubuntu subsystem (kernel name: Linux 4.4.0-18362-Microsoft).

2.2 Content

Two python files:

1. `lexicon.py` (containing a class `Lexicon` that implements the preprocessing and indexing modules);
2. `search.py` (containing a class `Search` that implements the boolean search and ranking modules).

Please check the usage of above files in `README.md`.

2.3 Preprocessing

To preprocess a text we need the following steps:

1. Tokenization;
2. Casefolding;
3. Stop-word checking;
4. Stemming.

For Tokenization, I simply used the package `re` to extract all the sequences of word characters [1] as tokens:

```
def tokenize(text):
    return re.findall(r'\w+', text)
```

In this way, all the letters and numbers (prevalent in a query) are preserved while punctuations are discarded. Next, lowercase the tokens using the built-in `token.lower()`, then check whether or not it is a stop word (using the provided stop word list [2]). Finally, use the Porter Stemmer from the package **nltk** to stem each token.

2.4 Indexing

The **positional inverted index** is essential for a document retrieval system because it allows us to do fast full text searches and it is easy to develop. However, it also has some disadvantages that cannot be neglected such as large storage, increased preprocessing and expensive maintenance.

The data structure I used for implementing inverted index is `OrderedDict` [3] from the package `collections`, the keys are the preprocessed tokens extracted from the corpus `trec.5000.xml`, for each token, the corresponding value is an `OrderedDict` of which the keys are document IDs and the values are positions of the token in each relevant document.

[1] A word character is a character from a-z, A-Z, 0-9, including the _ (underscore) character.
[2] Available at: http://members.unine.ch/jacques.savoy/clef/englishST.txt
[3] dict subclass that remembers the order entries were added

```
def pos_inv_ind(self):
    """build the positional inverted index"""
    for i, dp in self.docs_df.iterrows():
        doc = dp['doc']
        tokens = self.preprocess(doc)
        for pos in range(len(tokens)):
            """term:
                    docID: pos1, pos2, ...
                    docID: pos1, pos2, ...
            """
            self.index[tokens[pos]][int(i)] += [pos + 1]  # position starts from 1
    # sort the index by keys
    self.index = OrderedDict(sorted(self.index.items()))
```

The code for constructing the inverted index is rather simple. I stored the documents and their IDs in `dataFrame` from the package `pandas` in advance. For each row, preprocess the document (which consists of 'HEADLINE' and 'TEXT' sections in the original XML file) and get the tokens. Finally, update the inverted index by appending the position of each token in this document. Albeit the algorithm's simplicity, the time complexity is $O(nd)$ where n is the number of documents, d is the number of tokens in a document, suggesting that it is computationally expensive. Moreover, even with preprocessing, the resulting vocab size is 39308 given 5000 input documents. Although by Heap's Law, the number of new terms noticed will reduce over time while going through the documents, the number of the positions grows much faster. Consequently, the output `index.pkl` is of 11.0MB whereas the original corpus is of only 13.4MB.

The overall execution time (including preprocessing and indexing) is as follows:

```
real    0m53.640s
user    0m41.063s
sys     0m1.859s
```

Admittedly, it is not very efficient. But given the time complexity, it is reasonable.

2.5 Boolean Search

The implementation of the boolean search is split into the following:

1. Tokenizer:
   (a) Obtain tokens (operators and operands) from the query.
2. Operator-precedence Parser:
   (a) Encode the infix expression [4] (the tokenized query) into the postfix one [5];
   (b) Decode the postfix expression and evaluate the final result.
3. Singleton Search:
   (a) A single word;
   (b) A phrase (two consecutive words, the order **matters**);
   (c) A proximal pair (two words within certain distance, the order **does not matter**).

2.5.1 Tokenizer

There is a primary assumption of this tokenizer, i.e. when the bracket is used for specifying the precedence, it is treated as an operator and separated from the operand, e.g.

```
Text AND ( Technology OR #5(Data,Science) )
```

As such, the brackets used for indicating precedence are **differentiated** from brackets in a proximity query.

With this assumption, the first step is simply splitting on the space character. After doing so, only the proximity and the phrasal operands might be broken because they contain a space character. By tracking the start and the end symbol of such operands, we can combine the fractions as a whole, thus finishing the tokenization.

[4] Operators are written in-between their operands
[5] Operators are written after their operands, also known as "Reverse Polish notation"

2.5.2 Operator-precedence Parser

After tokenizing the example in 2.5.1, we obtain:

`['Text', 'AND', '(', 'Technology', 'OR', '#5(Data,Science)', ')']`

The next step is to encode this sequence from the infix expression to the postfix one, i.e.

`['Text', 'Technology', 'Science', 'OR', 'AND']`

Finally, we decode the postfix expression by reading from left to right, when encountering an operator, we evaluate the nearest operand(s) ('AND' and 'OR' require two operands whereas 'NOT' requires one). In our example here, this means we first read the three operands: `'Text'`, `'Technology'` and `'Science'`. Then we encounter `'OR'`, throw the nearest two operands `Technology` and `Science` into the Singleton Parser (see 2.5.3) and take the **union** of the retrieved documents (push it into the stack). Visually, it looks like:

`['Text', union('Technology', 'Science'), 'AND']`

Afterwards, we see `'AND'`, so we take the **intersection** of the retrieved documents of `'Text'` and `'union('Technology', 'Science')'`, thus obtaining the final result:

`[intersection('Text', union('Technology', 'Science'))]`

* For operator `'NOT'`, we take the **difference** of all documents minus the retrieved documents of the associate operand.
* The example above does not elaborate the preprocessing done on each token because it will be explained in 2.5.3.

We have seen a concrete example of the procedure of boolean search. Now, let us move on the detailed implementation.

[Shunting Yard Algorithm] [6]

```
/* This implementation does not implement composite functions,functions with variable number
     of arguments, and unary operators. */

while there are tokens to be read do:
    read a token.
    if the token is a number, then:
        push it to the output queue.
    if the token is a function then:
        push it onto the operator stack
    if the token is an operator, then:
        while ((there is a function at the top of the operator stack)
               or (there is an operator at the top of the operator stack with greater
    precedence)
               or (the operator at the top of the operator stack has equal precedence and is
     left associative))
              and (the operator at the top of the operator stack is not a left parenthesis):
            pop operators from the operator stack onto the output queue.
        push it onto the operator stack.
    if the token is a left paren (i.e. "("), then:
        push it onto the operator stack.
    if the token is a right paren (i.e. ")"), then:
        while the operator at the top of the operator stack is not a left paren:
            pop the operator from the operator stack onto the output queue.
        /* if the stack runs out without finding a left paren, then there are mismatched
    parentheses. */
        if there is a left paren at the top of the operator stack, then:
            pop the operator from the operator stack and discard it
after while loop, if operator stack not null, pop everything to output queue
if there are no more tokens to read then:
    while there are still operator tokens on the stack:
        /* if the operator token on the top of the stack is a paren, then there are
    mismatched parentheses. */
        pop the operator from the operator stack onto the output queue.
exit.
```

The **Shunting Yard Algorithm** is used for transferring infix expression to the postfix one. It has three main components:

1. Precedence:

[6] Pseudo code taken from Wikipedia: https://en.wikipedia.org/wiki/Shunting-yard_algorithm

(a) In our case, `'NOT' > 'AND' > 'OR' > '(' = ')'`.

2. Output Queue:
   (a) Append the token if it is an operand;
   (b) Pop the operator from the `Operator Stack` and append it into the `Output Queue` when the current operator has lower precedence than the popped one.
3. Operator Stack:
   (a) Append the operator if it has higher precedence than the first one in the stack, otherwise pop the first one in the stack;
   (b) Append the operator if it is a left bracket`'('`;
   (c) Pop the operator(s) when seeing a right bracket`')'` until matching a `'('`.

The key point of this algorithm is that operators of the lower precedence are like the disjunction points, e.g. in $5 + 3 \times 4$, $+$ separates 5 and 3×4.

2.5.3 Singleton Search

The singleton query includes: (a) single word; (b) phrase; (c) proximal pair. The words in such query must be preprocessed as in 2.3. For (a), simply return IDs of the documents that contain the word; For (b) and (c), we need to implement the **Linear Merge Algorithm** for better efficiency.

[Linear Merge Algorithm]
In the case of (b) or (c), we have two words, say `w1` and `w2`. For each of them, we first retrieve the sorted posting dict, i.e. `index[w1]` and `index[w2]` where `index` is the inverted index as in 2.4. Now, create two pointers `p1` and `p2`, we start our search from the smallest document ID in the keys of `index[w1]` and `index[w2]`. We compare the document IDs pointed by `p1` and `p2` and **always move the pointer that points to the smaller document ID**. If we find a match of the document IDs, compute the distances by:

```
dists = [j - i for i in index[w1][docID] for j in index[w2][docID]]
```

For the phrasal search, the order matters, so only when `dists` **contains** 1 will the document be retrieved; For the proximity search, the order does not matter, so it takes the absolute values of the distances and retrieve the document if **any absolute distance** is **smaller than or equal to** the maximum distance.

The execution time of running my boolean search on the given 10 queries is as follows:

```
real    0m2.272s
user    0m1.344s
sys     0m0.875s
```

Since both the Shunting Yard and the Linear Merge algorithms are of time complexity $O(n)$, the efficiency of the code has been guaranteed.

2.6 Ranking

The ranking system aims at rendering a ranked list of documents with the associated scores. In this project, the scores are computed based on a variant of **TF-IDF**. The weight of a term t in a document d is computed as:

$$\mathbf{w}(t,d) = (1 + log_{10}(\mathbf{tf}(t,d))) \times log_{10}(\frac{N}{\mathbf{df}(t)})$$

where **tf** is the term frequency of t in d, and **df** is the document frequency of t.

```
def _tf(self, term, docID):
    return len(self.index[term][docID])

def _df(self, term):
    return len(self.index[term].keys())
```

And the score of a query q in document d is computed as:

$$\mathbf{s}(q,d) = \sum_{t \in q, t \in d} \mathbf{w}(t,d)$$

Notice that only when the term appears both in the query and the document will the weight be calculated because otherwise **tf** would be zero.

Above are all about translating mathematical expression into code. The remaining question is: how should we choose the set of documents? The simplest method is to traverse all the document IDs, but in my implementation, I take the **union** of the documents that contain one of the tokens in the query. Therefore, the searching domain shrinks and it promotes efficiency. Finally, I used an `OrderedDict` to store the ranked documents and the associated scores.

The execution time of running the ranking on the given 10 queries is as follows:

```
real    0m3.543s
user    0m2.844s
sys     0m0.656s
```

It is slower than the boolean search but still very fast.

3 Comments

1. Both the efficiency (execution time) and the effectiveness (accurate documents retrieval) are carefully considered in this IR system.
2. Overall, I think the implementation of the boolean search is the most difficult and contains the most details. The Shunting Yard Algorithm is new to me and I spent hours to interpret it and transfer the pseudo code to the customized execution code. And for the Linear Merge Algorithm, there are many marginal cases needed to be considered. But on the bright side, such implementation enables my search to be more general than required, i.e. it can deal with queries containing multiple `'AND'`, `'OR'` and `'NOT'`.
3. Still have no idea that how to accelerate the creation of the inverted index.
4. At first, my implementation of the ranking system was extremely slow. Later on, I found that I accidentally traversed the whole vocab space! By changing one line of the code, the execution time reduces dramatically.

4 Future Work

1. **Generalize** the proximity and the phrasal searches by considering cases of more than two words, or even embed the boolean operators inside;
2. Make the inverted index **dynamic** meaning that it can be updated given new data;
3. Implement the Vector Space Model for the ranking system.