Cmpe 493 Assignment2 - Document Retrieval System

Emre Girgin - 2016400099

1. **Preprocessing Steps:**
   - **a.** Each *sgm* file in the folder is iterated.
   - **b.** In each file, each article is splitted from each other then parsing started:
     - **i.** Firstly, split for the "<DATE>" tag to get article id.
     - **ii.** Secondly, check if the "<TITLE>" and "<BODY>" tags exist. If both of them is absent, then skip that article.
     - **iii.** Lastly, split for the "<TITLE>" and "<BODY>" tag respectively, to get data to be processed.
   - **c.** Store the article id, title, and body to a class called "Story". Story class is a helper class to store articles and normalize them.
   - **d.** Normalize each story:
     - **i.** Firstly, extract tokens from the title and body meaning that split those texts based on whitespace.
     - **ii.** Secondly, remove all the punctuations in tokens, by replacing them with whitespace. This creates two separate tokens if there is a punctuation.
     - **iii.** Thirdly, apply case folding by lowercasing every token in that story.
     - **iv.** Lastly, remove the stopwords regarding the file provided.

2. **Inverted Index:**
   - **a.** Iterate over all the normalized stories.
   - **b.** For each token in the story, append the story id of a token to a Python dictionary. (Keys: token, Value: List of story id)
   - **c.** After appending a new id, sort the list of ids of that token.
   - **d.** At the end, dump that dictionary to a JSON file called "inverted_index.json".

3. **Trie**
   - **a.** The trie structure consists of two classes called "node" and "trie".

b. node class represents the nodes in the trie storing that the char it represents, a list of the children of that node (they are also an instance of node class), the end of word flag, and a list of document ids that contains that word.

c. trie class only has one attribute called root and two functions called insert and search. root attribute represents the root of the trie as the name suggests.

- **i.** insert : This function takes a string representing the token that is wanted to be inserted, and a document id of the story calling the insert function. In this function, we start from the root of the trie as the "current node" and iterate over the characters of the string. If the children nodes of the current node contain the node holding the character of the current character, then switch to that child node. Else, add a new child node to current node with storing the current char and then switch to the new child node. Apply this procedure until the end of the string. At the end of the string, mark current node's end of the word flag as true and append the document id to the list of document ids of the current node. Then sort the new list.
- **ii.** search : This function takes a string representing the searched token. We iterate over this token by switching to the child node of the current node (starting from the root) until it is possible. If at a point, any of the children nodes does not hold the current character, then return false, indicating an unsuccessful search (any of the documents does not contain this token). Else, complete until the end and return true with the list of document ids of the current node. However, if the current node's end of word token is false when iteration is over, then return false again.
- **iii.** save : This function basically saves the trie into a pickle file.

d. Note that, since "eq" operator of the node class is overwritten, the "in" (a.k.a. contains) keyword has much easier use in both search and insert.

4. **Code**

```
import pickle

class node:

    def __init__(self,  char=None):
        self.char = char
        self.children = []
        self.end_of_word = False
        self.doc_ids = []

    def __repr__(self):
        return str(self.char)

    def __str__(self):
        return str(self.char)

    def __eq__(self, other):
        return str(self.char) == other

class trie:

    def __init__(self):
        self.root = node()



    def insert(self, string, doc_id):
        """
            The function inserting a new token into the trie and adding doc_id to its
field if the token is already exits.
            This function is called every time a token is processed. This means multiple
call for a token but different document ids.
            With every call of this function for a certain token, the document ids are
appended to a list, meaning that the list of document ids containing this token.
        """

        current_node = self.root

        for char in string:
```

```
            # Since the eq operator is overwritten we can directly check a char int a list
of nodes.
            if char in current_node.children:
                current_node = current_node.children[current_node.children.index(char)]
            else:
                current_node.children.append(node(char))
                current_node = current_node.children[current_node.children.index(char)]


        current_node.end_of_word = True

        current_node.doc_ids.append(int(doc_id))
        current_node.doc_ids.sort()

    def search(self, string):

        """
            If a token is already inserted to the trie, then this method returns True and
list of document ids of the documents containing that token.
            If token is absent, this function returns False and an empty list.

        """

        current_node = self.root


        for char in string:
            # Since the eq operator is overwritten we can directly check a char int a list
of nodes.
            if char in current_node.children:
                current_node = current_node.children[current_node.children.index(char)]
            else:
                # Does not exist in the trie.
                return False, []


        if current_node.end_of_word:
            return True, current_node.doc_ids
        else:
            return False, []
```

```
def save_trie(self):
    with open("trie.pickle", "ab") as pckFile:
        pickle.dump(self, pckFile)
```