Mahir Efe KAYA
2016400195

CMPE 493

Assignment 2 Report

Description: Create a simple search engine which will get a query with or without wildcard and find the indexes of its occurences.

Solution:

a) Firstly I had to seperate the sections of:

*Body

*NEWID

*Title

to get the needed text from them. I used string.find to find seperate articles by finding their starting point and ending point. Then I used the same thing to seperate other needed sections. Substringing was quite easy in python, I didn't have any problems with that.

Then I had to normalize the words. The normalization we were requested were: remove punctuations from the texts, do the casefolding and removing the stop words from the list of the words. However removing punctuations would prevent me from splitting the articles. So I kept removal of punctuations after the separations of the text, body, id and title.

To casefold was easy. Only one line of the code do the trick.

To remove the stop words, first I had removed the duplicates, so unnecessary traversing would not be done in removal searches.

b) To store inverted index list, I used python dictionary that words are the key set, and indexes are in the list keys refer to.

After getting the words from one article, I added them to that python dictionary, or appended the existent ones with the current article's id.

c) I created two new classes for the trie and its nodes. Nodes only holds the values of the Boolean isEnd and children dictionary. Children holds the characters as key, the children nodes as value and isEnd tells us if this far is a word.

Construction:

Mahir Efe KAYA
2016400195

In trie, I configured one node to be root, so added words can be connected to it. To add, firstly add function is called in which a function called addSequence is called. addSequence where the real magic happens. Firstly, children nodes of the current node is retrieved then checked if the character we want to add does exist in the list. If not, a new node is added, then continued from that node. If so, without adding any new node, continues from that node. Once the length is equal to the length of the string we want to add, function stops.

Get:

Trie.get is mostly for query processing. There are also two functions in sequence for this function too. Get only gets a pattern string, then calls for the getSequence in which the real magic happens. In this function, several parameters is used in a recurrence fashion:

-pattern which is retrieved from the user,

-curr : node, which determines the current we are on

-sofar: str, for storing the string collected so far

-length, which determines the stage we're on the pattern.

pattern is traversed character by character to traverse on the trie. There is a certain check for pattern's characters if it is wildcard or not.

If it is a wildcard, for to be ascertain that every case is evaluated, I had to add the situations where wildcard does not equal to any sequence of characters.

```
if pattern[length] == '*':
    merge = self.getSequence(pattern, curr, sofar, length+1)
    for char in children:
        node = children[char]
        merge += self.getSequence(pattern, node, sofar+char, length)
    return merge
```

For the case of wildcard mean nothing, I just increased the length without adding anything to sofar to pass the wildcard character.

For the case it might mean something, I call the function for the every child's value. Then merge them all into one list to return.

If it is not a wildcard, the function is call with the character is added to the sofar value and length incremented by one.

```
        else:
            if pattern[length] in children:
                strlist = self.getSequence(pattern, children[pattern[length]],
                                           sofar + pattern[length], length+1)
                return strlist
            else:
                return []
```

In the end, if length is equal to the length of the pattern the function stop recursion and start returning values to the back.

Querry processor is only for retrieving the data from ‘trie.pickle’ and invertedindex.json, and calling trie.get function with the user input, then retrieving the indexes from the invertedindex dictionary.

```
from node import node
class trie:
    def __init__(self):
        self.root = node() #create a node for to be root of traditional trie structure

    def add(self,string : str): #starts the add sequence
        self.addSequence(self.root,string,0)
    def addSequence(self,curr : node,string : str,length : int): #traditional algorithm for adding to a trie
        if len(string) == length: #checks if the end of the word is reached and marks it
            curr.isWord = True
            return
        nodes = curr.get_children()
        if string[length] in curr.get_children() : #checks if the character to be added at that location

            self.addSequence(nodes[string[length]],string,length+1)
        else: #if it does not exist in children adds it
            curr.get_children()[string[length]] = node()
            self.addSequence(nodes[string[length]],string,length+1)

    def get(self,pattern : str): # starts the get sequence
        return list(set(self.getSequence(pattern,self.root, "", 0)))

    def getSequence(self,pattern : str, curr : node, sofar : str,length : int):

        if curr == None: #checks the current node for errorous input
            return []
        children = curr.get_children()
        if len(pattern) == length:  #checks if the end of the pattern has already been reached
            if curr.isWord:         #if it is a word, returns it
                return [sofar]
            return []

        if pattern[length] == '*': #checks the wildcards
            #it calls a case where wildcard amount to nothing
            merge = self.getSequence(pattern, curr, sofar, length+1)
            #calls a function for every child to cover all the cases and merges their results
            for char in children:
                node = children[char]
                merge += self.getSequence(pattern, node, sofar+char, length)
            return merge
        else:
            #if it is a normal character in the pattern, it tries to find the
            #char in children, if it cannot, it returns empty list
            if pattern[length] in children:
                strlist = self.getSequence(pattern, children[pattern[length]],
                                           sofar + pattern[length], length+1)
                return strlist
            else:
                return []
```