Lab Assignment 4

ISMAEL VILLALOBS
PROFESSOR FUENTES
TA: DITA NATH
10-23-19

Introduction

We were prompted to compare the running times of two data structures used to retrieve word embeddings to enable the (hopefully fast) comparison of two given words. One of your table implementations should use a binary search tree; the other should use a B-tree. We will use the text file from Stanford's GloVe project, to determine similarities between words. The analysis of the running time for building each of the trees as well as performing several operations to manipulate or retrieve data from each of these structures is crucial to determine the advantages of each in further real-world applications.

Proposed Solutions

Part 1- This portion of the lab we had to prompt the user to choose between Binary Search Tree implementation, B-Tree implementation or an option to end. If the B-Tree option is chosen the user is prompted to enter the maximum number of items to store in a node followed by printing the stats of the tree such as height, number of nodes, and time taken to build. Similar to the B-Tree if the user prompted to use a Binary Search Tree, it would make the tree from the text file as well as the stats of the tree such as height, number of items, and the time elapsed to build BST.

Part 2- For this part we had to build either a Binary Search Tree or a B-Tree with the words from the file "glove.6B.50d.txt". In either tree implementation, each line of the file is read, being tokenized into a list of strings, for the values found on that line. Now, looking at the data from the glove file, each of the words in the file is to be stored as a word in a WordEmbedding object, and the embeddings that followed as a list in the WordEmbedding object. Well, the word is always the first string in the line. Therefore, my implementation checks to make sure that the first string in the list of tokens for a line begins with a letter. If that's the case, the WordEmbedding object is inserted into the BST. In both implementations, once the file is traversed completely, the file is closed and the built tree implementation is returned to be used further.

Part 3 – This portion asks us to find the "Similarity " of words. For this, we had to read in a second file that contained pairs of words and for these words we had to compute the similarities of the words. To compute the similarity between words w0 and w1, with embeddings e0 and e1, we use the cosine distance, which ranges from -1 (very different) to 1 (very similar), given by: $sim(w0, w1) = e0 \cdot e1 / |e0||e1|$

Experimental Results

Binary Search Tree

```
Choose an implementation: 1
Creating Binary Search Tree!
Please be patient its alot of words!

Running time for constructing BST 15.071  seconds
Height:  52
Number of items:  400000
Similarity [ bear bear ] = 1.000
Similarity [ barley shrimp ] = 0.535
Similarity [ barley oat ] = 0.670
Similarity [ federer baseball ] = 0.287
Similarity [ federer tennis ] = 0.717
Similarity [ harvard stanford ] = 0.847
Similarity [ harvard utep ] = 0.068
Similarity [ harvard ant ] = -0.027
Similarity [ raven crow ] = 0.615
Similarity [ raven whale ] = 0.329
Similarity [ spain france ] = 0.791
Similarity [ spain mexico ] = 0.751
Similarity [ mexico france ] = 0.548
Similarity [ mexico guatemala ] = 0.811
Similarity [ computer platypus ] = -0.128

Time taken to compute similarities 0.016  seconds
```

B-Tree

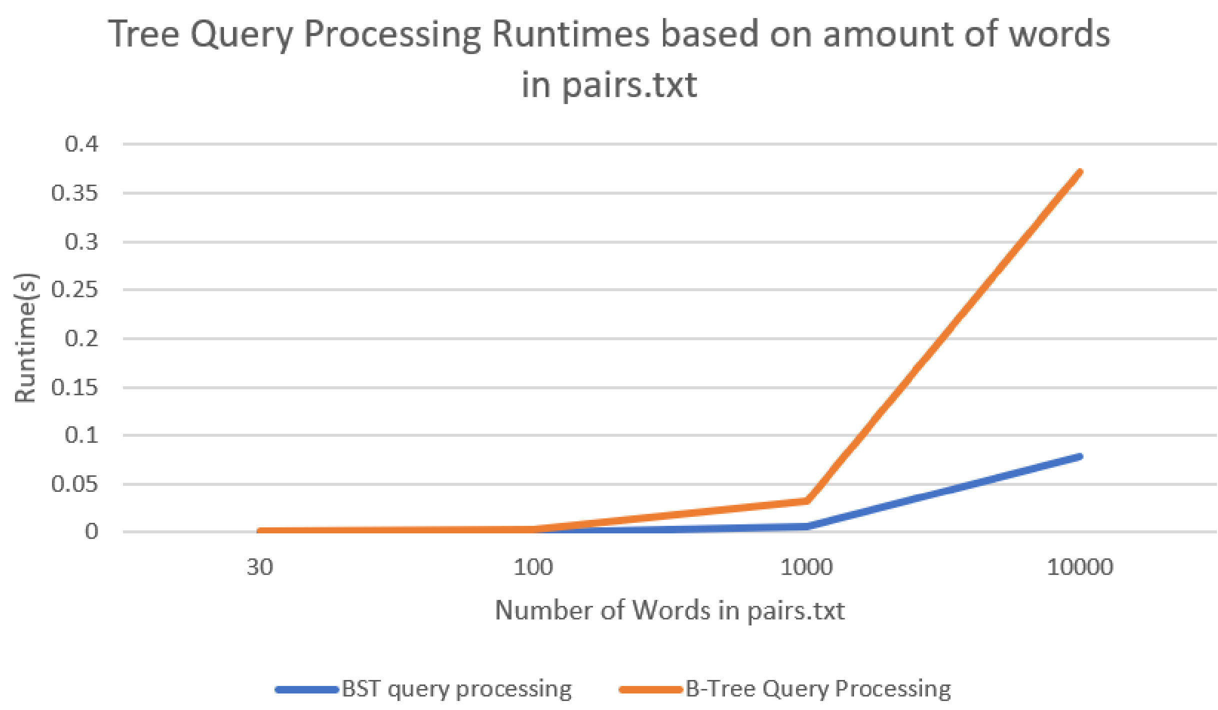
```
Choose an implementation: 2

Enter max data: 5
Creating B-Tree!
Please be patient its alot of words!
Running time for constructing BTree 16.829  seconds
Height:  9
Number of items:  400000

Similarity [ bear bear ] = 1.000
Similarity [ barley shrimp ] = 0.535
Similarity [ barley oat ] = 0.670
Similarity [ federer baseball ] = 0.287
Similarity [ federer tennis ] = 0.717
Similarity [ harvard stanford ] = 0.847
Similarity [ harvard utep ] = 0.068
Similarity [ harvard ant ] = -0.027
Similarity [ raven crow ] = 0.615
Similarity [ raven whale ] = 0.329
Similarity [ spain france ] = 0.791
Similarity [ spain mexico ] = 0.751
Similarity [ mexico france ] = 0.548
Similarity [ mexico guatemala ] = 0.811
Similarity [ computer platypus ] = -0.128

Time taken to compute similarities 0.016  seconds
```

Big O Notation & Runtimes



Conclusion

As seen by my data B-Tree Implementation as was expected. While the B-tree implementation generally has less nodes than the binary search tree implementation (See beginning of Experimental results for screenshots of output), the BST implementation has better efficiency when it comes to building the tree, populating it with word embeddings. Remember that, the data each node can store up to max_data values. As the max number of items that can be stored per node in a B-Tree increases, the runtimes begin to even out and are the same as evidenced by the table

Academic Honesty

"I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class."

Name: Ismael Villalobos

Appendix:

'''

```
Ismael Villalobos

10-23-19

Lab Assignment 4

Professor Fuentes

TA-Dita Nath

Purpose:
'''
import numpy as np


import time
class BST(object):
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right


def NumberOfNodes(T):
    if T is None:
        return 0
    leftNum = NumberOfNodes(T.left)
    rightNum = NumberOfNodes(T.right)

    return 1 + sum([leftNum,rightNum])


def Height(T):
    if T == None:
        return -1
    lh = Height(T.left)
    rh = Height(T.right)
    return 1+ max([lh,rh])
```

```
def PrintInOrder(T):
   if T is not None:
      PrintInOrder(T.left)
      print(T.data.word, T.data.emb)
      PrintInOrder(T.right)



def InsertBST(T,newWord):
   if T == None:
      T = BST(newWord)
   elif T.data.word > newWord.word:
      T.left = InsertBST(T.left,newWord)
   else:
      T.right = InsertBST(T.right,newWord)
   return T


def SearchBST(T, k):
   if T is None or T.data.word ==k:
      return T
   elif k<T.data.word:
      return SearchBST(T.left,k)


   else:
      return SearchBST(T.right, k)
class BTree(object):
   # Constructor
   def __init__(self,data,child=[],isLeaf=True,max_data=5):
      self.data = data
```

```
        self.child = child
        self.isLeaf = isLeaf
        if max_data <3: #max_data must be odd and greater or equal to 3
            max_data = 3
        if max_data%2 == 0: #max_data must be odd and greater or equal to 3
            max_data +=1
        self.max_data = max_data


def FindChild(T,k):
    # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
    for i in range(len(T.data)):

        if k.word < T.data[i].word:
            return i
    return len(T.data)


def InsertInternal(T,wordObj):
    # T cannot be Full
    if T.isLeaf:
        InsertLeaf(T,wordObj)
    else:
        k = FindChild(T,wordObj)
        if IsFull(T.child[k]):
            m, l, r = Split(T.child[k])
            T.data.insert(k,m)
            T.child[k] = l
            T.child.insert(k+1,r)
            k = FindChild(T,wordObj)
```

```
        InsertInternal(T.child[k],wordObj)

def Split(T):
    #print('Splitting')
    #PrintNode(T)
    mid = T.max_data//2
    if T.isLeaf:
        leftChild = BTree(T.data[:mid],max_data=T.max_data)
        rightChild = BTree(T.data[mid+1:],max_data=T.max_data)
    else:
        leftChild = BTree(T.data[:mid],T.child[:mid+1],T.isLeaf,max_data=T.max_data)
        rightChild = BTree(T.data[mid+1:],T.child[mid+1:],T.isLeaf,max_data=T.max_data)
    return T.data[mid], leftChild,  rightChild

def InsertLeaf(T,i):
    #appends data to node
    T.data.append(i)

    #sorting in alphabetical order
    T.data.sort(key = lambda x: x.word)

def IsFull(T):
    return len(T.data) >= T.max_data

def Leaves(T):
    # Returns the leaves in a b-tree
    if T.isLeaf:
        return [T.data]
    s = []
    for c in T.child:
```

```
        s = s + Leaves(c)
    return s


def InsertBTree(T,i):
    if not IsFull(T):
        InsertInternal(T,i)
    else:
        m, l, r = Split(T)
        T.data =[m]
        T.child = [l,r]
        T.isLeaf = False
        k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def HeightBTree(T):
    if T.isLeaf:
        return 0
    return 1 + HeightBTree(T.child[0])

def Print(T):
    # Prints data in tree in ascending order
    if T.isLeaf:
        for t in T.data:
            print(t,end=' ')
    else:
        for i in range(len(T.data)):
            Print(T.child[i])
            print(T.data[i],end=' ')
        Print(T.child[len(T.data)])
```

```
#returns number of nodes.
def NumberOfItems(T):
    sum = len(T.data)
    #iterating through each node of each child
    for i in T.child:
        sum+=NumberOfItems(i)

    return sum

def PrintD(T,space):
    # Prints data and structure of B-tree
    if T.isLeaf:
        for i in range(len(T.data)-1,-1,-1):
            print(space,T.data[i])
    else:
        PrintD(T.child[len(T.data)],space+'   ')
        for i in range(len(T.data)-1,-1,-1):
            print(space,T.data[i])
            PrintD(T.child[i],space+'   ')



def SearchBTree(T,k):

    for i in range(len( T.data)):
        if k.word == T.data[i].word:
            return T.data[i]
        if T.isLeaf:
            return None
        return SearchBTree(T.child[FindChild(T,k)],k)
```

```
class WordEmbedding(object):
        def __init__(self,word,embedding=[]):
                # word must be a string, embedding can be a list or and array of ints or floats
                self.word = word
                self.emb = np.array(embedding, dtype=np.float32) # For Lab 4, len(embedding=50)



if __name__ =="__main__":

  while True:
    print('\n1. BST Implementation \n2. B-Tree Implementaion \n3. Exit ')

    choice  = int(input('Choose an implementation: '))


    if choice==1:
      BinaryST=None

      print('Creating Binary Search Tree!')
      print('Please be patient its alot of words!')
      print()
      with open("glove.6B.50d.txt",'r',encoding='utf-8') as f:
        start = time.time()
        for line in f:
          bin_list=line.split(" ")
          word_object=WordEmbedding(bin_list[0],bin_list[1:])
```

```
        BinaryST= InsertBST(BinaryST,word_object)
    end = time.time()
    print('Running time for constructing BST', "{:.3f}".format(end-start),' seconds')
    print('Height: ', Height(BinaryST))
    print('Number of items: ',NumberOfNodes(BinaryST))


    with open("pairs.txt","r") as f2:
      start2= time.time()
      for line2 in f2:
        listBST= line2.split(" ")


        listBST[1] = listBST[1].strip()


        word1 = SearchBST(BinaryST,listBST[0])
        word2 = SearchBST(BinaryST,listBST[1])


        distance =
np.dot(word1.data.emb,word2.data.emb)/(abs(np.linalg.norm(word1.data.emb))*abs(np.linalg.norm(w
ord2.data.emb)))


        print("Similarity [", word1.data.word, word2.data.word, "] =","{:.3f}".format(distance))



    end2 = time.time()
    print("\nTime taken to compute similarities", "{:.3f}".format(end2-start2),' seconds',"\n")



  if choice==2:
    input_max_data = int(input('Enter max data: '))


    T = BTree([],max_data =input_max_data )
```

```
print('Creating B-Tree!')
print('Please be patient its alot of words!')

with open("glove.6B.50d.txt",'r',encoding='utf-8') as file:
    start = time.time()
    for line in file:
        eachline=line.split(" ")

        word_object = WordEmbedding(eachline[0],eachline[1:])

        InsertBTree(T,word_object)
end = time.time()
print('Running time for constructing BTree',"{:.3f}".format(end-start),' seconds')
print('Height: ', HeightBTree(T))
print('Number of items: ',NumberOfItems(T))

with open("pairs.txt",'r') as file2:
    start1 = time.time()

    for line in file2:
        line = line.strip().split(" ")

        #searching for each word in B-tree
        obj1 = WordEmbedding(line[0]) #creating an object with the current word
        obj2 = WordEmbedding(line[1])

        word1 = SearchBTree(T,obj1) #searching for the first word
        word2 = SearchBTree(T,obj2) #searching for second word
```

```
            distance =
np.dot(word1.emb,word2.emb)/(abs(np.linalg.norm(word1.emb))*abs(np.linalg.norm(word2.emb)))

            print("Similarity [", word1.word, word2.word, "] =","{:.3f}".format(distance))

        end2 = time.time()
        print("\nTime taken to compute similarities","{:.3f}".format(end2-start2),' seconds',"\n")

    elif choice ==3:
        print('Ending program, BYE!')
        break
```