Laurence Labayen

80358755

10/21/2019

Lab #4

CS2302 MW 10:30am

Description:

For this lab, we were given the task of implementing a list of words and its corresponding embedding into a B-Tree and a Binary Search Tree. A word file containing 30000+ word embeddings was given to us from the Natural Language Processing website. With the file, we were asked to put each word and embedding into a Node then insert them into a B-Tree or Binary Search Tree. Then find the height, number of nodes, and running time of building the corresponding tree. After, we use pairs of words from another word file and find its similarities using the embedding of each given word and the cosine distance.

"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:

$$sim(w0,w1) = e0 \cdot e1 \; |e0| \, ||e1|$$

where e0 · e1 is the dot product of e0 and e1 and |e0| and |e1| are the magnitudes of e0 and e1.
Recall that the dot product of vectors u and v of length n is given by $u \cdot v = u0 * v0 + u1 * v1 + \ldots + un-1 * vn-1$
(you can use np.dot(u,v)) and the magnitude of a vector u of length n is given by $|u| = \sqrt{u \cdot u} = u20 + u21 + \ldots u2n-1$ *(you can use np.linalg.norm(u)).*"

Solution design and Implementation:

Creating constructors:

I created a constructor for a Node that will hold the word and its corresponding embedding. Next was a standard constructor for a B-Tree that takes a data list ,child list, and max data. Lastly, the Binary Search Tree constructor takes some data, left child and right child.

```
#Word embedding node constructor used to store a
#word and the corresponding embedding
class WE_Node(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)

#B-Tree constructor used to store data, child, isLeaf and max_data
class BTree(object):
    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
# Binary Search Tree constructor
class BST(object):
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

Creating Trees from file:

The approach that i took was to open the GloVe file and read the lines. I added a check where it only allows alphabetic characters to remove unwanted words/strings into the tree. The first row was always the word so I implemented a loop to pick the first row and put that into a node and the rest was its embedding. Inside the loop, I added each word embedding Node into the corresponding tree using a modified Insert function to sort the tree alphabetically using the word in each node. During all of which, a timer was ran to get the running time of building the corresponding tree.

```
    # Open glove file
    file = open('glove.6B.50d.txt','r')

    # Start counter
    start = time.perf_counter()

    # Read file line by line
    for line in file.readlines():
         row = line.strip().split(' ')
         # Check if word matches the pattern of characters
         if pattern.fullmatch(row[0]) is not None:
             # Insert into B-Tree with word and its embedding
             InsertBTree(T,WE_Node(row[0],[(i) for i in row[1:]]))
    # Stop counter
    end = time.perf_counter()
```

Modified functions:

To get the code working correctly, I modified some of the functions to look at the word inside the node

```
def InsertLeaf(T,i):
    T.data.append(i)
    #sorts the current T.data based on the word in the WE_Node
    T.data.sort(key=lambda x: x.word)

def FindChild(T,k):
    #checks if k is an instance of WE_Node
    if isinstance(k, WE_Node):
        for i in range(len(T.data)):
            #if k is a WE_Node, access the word in the Node using k.word
            #if k is less than the value of the current word, return the current
            #postiion
            if k.word < T.data[i].word:
                return i
    #otherwise, k is a string that can be compared to T.data[i].word
    else:
        for i in range(len(T.data)):
            if k < T.data[i].word:
                return i
    #if the loop ends, return the length of current T.data to point to last child
    return len(T.data)
```

instead of the Node itself. For example, the InsertLeaf function was modified to sort the list using the word inside each node.

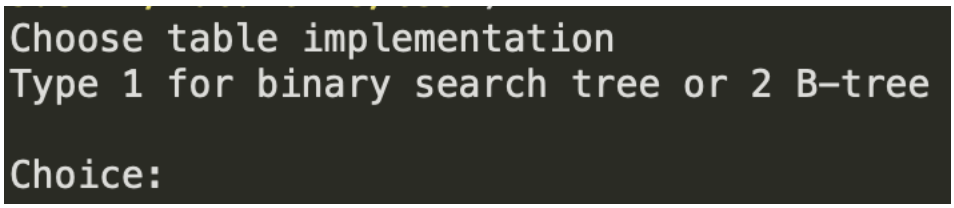
Similarities:

First, I had to create a file with 300 pairs of words. Then read the file to a list of lists, with the pairs in each list. From there, I created a loop to iterate through the list of pairs and pick out the first and second word. Using each word as an argument to search the corresponding tree and find the Node that matches

(checking in between it returns None if the Node/word was not found). After receiving the returned Node, I grab the embedding side of it to be used to calculate the similarities using the cosine distance formula. Inside the loop, I timed each iteration (not including printing) and added each timed iteration to a variable to get the total time of searching and calculating without the printing.

```
296 def Similarity(T,choice, file_choice, num_pairs):
297     # Open and read pairs word file and insert into a list as list of pairs
298     pairs=[line.strip().split(' ') for line in open(file_choice,'r')]
299 
300     # Assign timer to 0
301     timer=0
302 
303     # Loop to iterate through list of pairs line by line
304     for i in range(num_pairs):
305         # Start timer
306         start = time.perf_counter()
307 
308         # word1 gets the first column in each line from pairs list
309         word1=pairs[i][0]
310         # word2 gets the second column in each line from pairs list
311         word2=pairs[i][1]
312 
313         #word1emb and word2emb gets the embedding that is found by
314         #using SearchBST (for BST) or SearchBTree (for B-Tree)
315         word1emb=(choice(T,word1))
316         word2emb=(choice(T,word2))
317 
318         # Check if word1emb or word2emb is not found
319         if word1emb is None or word2emb is None:
320             continue
321 
322         # Formula to find cosine distance between both word embeddings
323         cosine_distance = np.dot(word1emb.emb, word2emb.emb)/(np.linalg.norm(word1emb.em
324         # Stop timer for every iteration
325         end = time.perf_counter()
326 
327         # Add each timed iteration of finding similaties to "timer"
328         timer += end - start
329 
330         print(i+1,'Similarity ['+word1+','+word2+'] =',str(round(100*cosine_distance,5))
```

Experimental Results:



Binary Search Tree construction running time: 23.24 seconds

```
Choice: 1
Loading glove file...
Loaded glove file
Building Binary Search Tree...

Binary Search Tree stats:

Number of nodes: 317756
Height: 48
Running time for Binary Search Tree construction: 23.239739
```

B-Tree construction running times:

max_items=5: 29.82 seconds

```
Choice: 2

Maximum number of items in node: 5
Loading glove file...
Loaded glove file
Building B-tree...

B-tree stats:

Number of nodes: 317756
Height: 9
Running time for B-tree construction (with max_items = 5): 29.815724
```

max_items=11: 32.27669 seconds

```
Choice: 2
Loading glove file...
Loaded glove file
Building B-tree...

B-tree stats:

Number of nodes: 317756
Height: 5
Running time for B-tree construction (with max_items = 11): 32.27669
```

max_items=3: 34.944583 seconds

```
Maximum number of items in node: 3
Loading glove file...
Loaded glove file
Building B-tree...

B-tree stats:

Number of nodes: 317756
Height: 13
Running time for B-tree construction (with max_items = 3): 34.944583
```

Binary Search Tree search with 300 pairs:

```
294 Similarity [logical,reasoning] = 79.4257%
295 Similarity [moral,ethics] = 66.82629%
296 Similarity [psychology,sociology] = 89.05489%
297 Similarity [statistics,numbers] = 66.82225%
298 Similarity [history,world] = 70.91538%
299 Similarity [digital,analog] = 78.40965%
300 Similarity [computer,platypus] = -12.76999%


Running time for similarities with 300 pairs: 0.0271
```

B-Tree search with 300 pairs (max_items=5):

```
298 Similarity [history,world] = 70.91538%
299 Similarity [digital,analog] = 78.40965%
300 Similarity [computer,platypus] = -12.76999%


Running time for similarities with 300 pairs: 0.0655
```

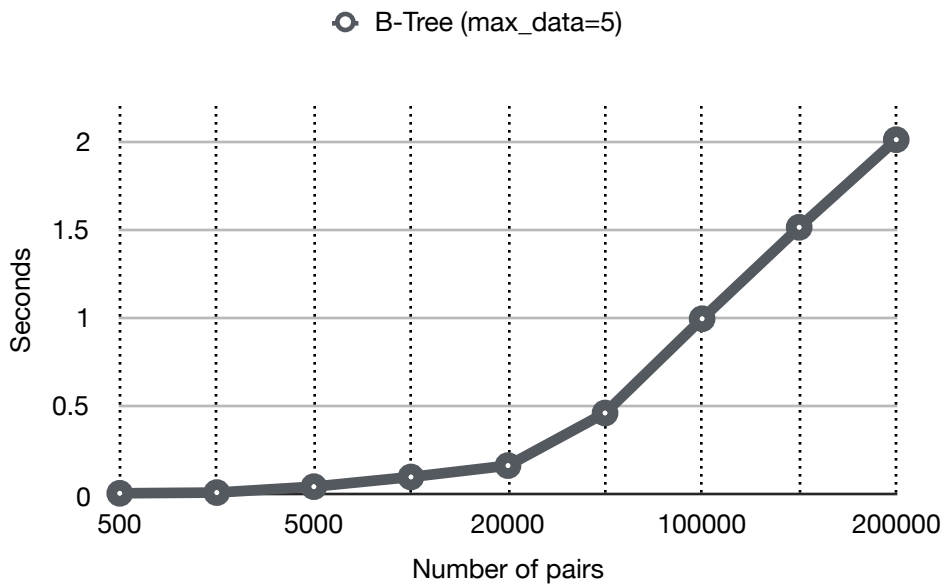
B-Tree search with 300 pairs (max_items=11):

```
Running time for similarities with 300 pairs: 0.0353

Maximum number of items in node: 11
```

B-Tree search with 300 pairs (max_items=3):

```
Running time for similarities with 300 pairs: 0.0355
```

B-Tree search and calculate similarity running times:



Max Data= 5

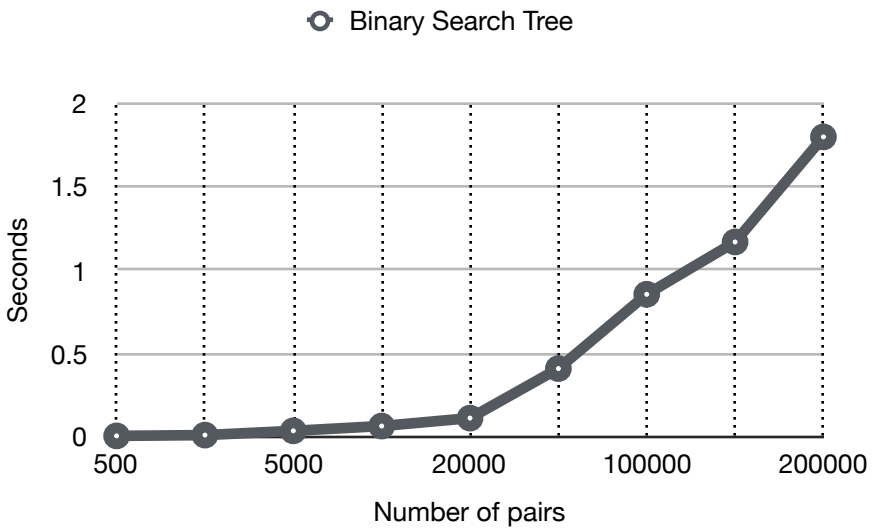
Elements	500	1000	5000	10000	20000	50000	100000	150000	200000
Time	0.0082	0.0123	0.0457	0.1003	0.1647	0.4617	0.9944	1.5111	2.006

Max Data = 3

Elements	500	1000	5000	10000	20000	50000	100000	150000	200000
Time	0.0076	0.0135	0.0549	0.111	0.1731	0.565	1.2809	1.8101	2.4178

```
Running time for similarities with 100000 pairs: 0.9944

Enter number of random pairs: 100000


Running time for similarities with 150000 pairs: 1.5111

Enter number of random pairs: 150000


Running time for similarities with 200000 pairs: 2.006

Enter number of random pairs: 200000
```

Binary Search Tree running times:



Elements	500	1000	5000	10000	20000	50000	100000	150000	200000
Time	0.0045	0.0092	0.0348	0.0636	0.1112	0.409	0.8554	1.1697	1.8011

```
Running time for similarities with 100000 pairs: 0.8554

Enter number of random pairs: 150000

Running time for similarities with 150000 pairs: 1.1697

Enter number of random pairs: 200000

Running time for similarities with 200000 pairs: 1.8011
```

Conclusion:

This lab gave me a better understanding of implementing B-Trees and Binary Search Trees in a much larger scale. Both trees have about the same running time efficiency. Even with changing the max items in the B-Tree, running times for searching seem to be the same. The only notable difference when changing the max items in the B-Tree is that the construction time. Additionally, I learned about word embeddings and their use for artificial intelligence. Word embeddings can be very useful for machine learning and prediction for data science.

Appendix:

```
import numpy as np
import re
import time

#Word embedding node constructor used to store a
#word and the corresponding embedding
class WE_Node(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)

#B-Tree constructor used to store data, child, isLeaf and max_data
class BTree(object):
    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

# Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
def FindChild(T,k):
    #checks if k is an instance of WE_Node
    if isinstance(k, WE_Node):
        for i in range(len(T.data)):
            #if k is a WE_Node, access the word in the Node using k.word
            #if k is less than the value of the current word, return the current
            #postiion
            if k.word < T.data[i].word:
                return i
    #otherwise, k is a string that can be compared to T.data[i].word
    else:
        for i in range(len(T.data)):
            if k < T.data[i].word:
                return i
    #if the loop ends, return the length of current T.data to point to last child
    return len(T.data)

def InsertInternal(T,i):
    if T.isLeaf:
        InsertLeaf(T,i)
    else:
        k = FindChild(T,i)
        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,i)
        InsertInternal(T.child[k],i)

# Split function is used when max data is reached
def Split(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):
    T.data.append(i)
    #sorts the current T.data based on the word in the WE_Node
    T.data.sort(key=lambda x: x.word)

# Check if current t.data is full
def IsFull(T):
    return len(T.data) >= T.max_data

# Inserts i to T
def InsertBTree(T,i):
    #check if T is not full
    if not IsFull(T):
        InsertInternal(T,i)
    #otherwise, use the split function and insert internally
    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)

# Gets height of T by recursively calling left child until T.isLeaf.
def Height(T):
    if T.isLeaf:
        return 0
    return 1 + Height(T.child[0])

# Prints data in tree in ascending order
def Print(T):
    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)])
```

```
# Prints data and structure of B-tree
def PrintD(T,space):
    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):
    # Returns node where k is, or None if k is not in the tree
    for i in range(len(T.data)):
        if k == T.data[i].word:
            return T.data[i]
    if T.isLeaf:
        return None
    return SearchBTree(T.child[FindChild(T,k)],k)

def NumItems(T):
    s = len(T.data)
    if not T.isLeaf:
        for i in range(len(T.child)):
            s+=NumItems(T.child[i])
    return s

#---------------------------------------------------------------------------
# Binary Search Tree constructor
class BST(object):
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
# Insert function
def InsertBST(T,newItem):
    if T == None:
        T =  BST(newItem)
    #check if newItem goes to T.left
    elif T.data.word > newItem.word:
        T.left = InsertBST(T.left,newItem)
    #otherwise, newItem goes to T.right
    else:
        T.right = InsertBST(T.right,newItem)
    return T

def PrintBST(T):
    if T is None:
        return
    Print(T.left)
    print(T.data)
    Print(T.right)
```

```
# Find function
def SearchBST(T, k):
    if T is not None:
        return _SearchBST(T, k)
    else:
        return None
# Find function helper
def _SearchBST(T, k):

    # base case. if k is equal to word in T.data, return T.data
    if T.data.word==k:
        return T.data
    # if k's value is greater than word in T.data, traverse left
    if k < T.data.word:
        return SearchBST(T.left, k)
    # otherwise, traverse left
    else:
        return SearchBST(T.right, k)

# Counts total nodes in the BST
def NodeCount(T):
    count=0
    if T != None:
        count+=1
        count= count + NodeCount(T.left)
        count= count + NodeCount(T.right)
    return count
# Returns the Height of the BST by traversing the entire tree
# and keeping count
def GetHeight(T):
    if T is None:
        return -1
    left_height = GetHeight(T.left)
    right_height = GetHeight(T.right)
    return 1+max(left_height, right_height)
#--------------------------------------------------------------------------

# Test function using a B-Tree
def BTree_Test():
    # Input for max data for B-Tree
    maxdata=int(input('Maximum number of items in node: '))

    # B-Tree assigment
    T = BTree([],[],max_data=maxdata)

    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')

    # Open glove file
    file = open('glove.6B.50d.txt','r')

    # Start counter
    start = time.perf_counter()
```

```
    # Read file line by line
    for line in file.readlines():
        row = line.strip().split(' ')
        # Check if word matches the pattern of characters
        if pattern.fullmatch(row[0]) is not None:
            # Insert into B-Tree with word and its embedding
            InsertBTree(T,WE_Node(row[0],[(i) for i in row[1:]]))
    # Stop counter
    end = time.perf_counter()
    print('Loaded glove file')
    file.close()


    print('Building B-tree...\n')

    print('B-tree stats:\n')
    print('Number of nodes:', NumItems(T))
    print('Height:', Height(T))
    print('Running time for B-tree construction (with max_items = '+str(maxdata)+'): '+
str(round((end - start), 6))+'\n')

    # Call Similarity function with T and "Search"
    Similarity(T, SearchBTree, 'pairs.txt', 300)

    # Similarity test for more words
    yn = input('Test similarities again with random words? Y/N ')
    if yn.lower() == 'y':
       num_pairs=int(input('Enter number of random pairs (up to 9999): '))
       if num_pairs <= 9999:
          Similarity(T, SearchBTree, 'pairs10000.txt', num_pairs)

    return T
def BST_Test():

    # Assign T
    T=None
    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')

    # Open glove file
    file = open('glove.6B.50d.txt','r')
    start = time.perf_counter()

    # Read file line by line
    for line in file.readlines():
        row = line.strip().split(' ')

        # Check if word matches the pattern of characters
        if pattern.fullmatch(row[0]) is not None:
            # Insert into BST with word and its embedding
            T=InsertBST(T, WE_Node(row[0],[(i) for i in row[1:]]))
    # Stop timer
```

```
    end = time.perf_counter()
    print('Loaded glove file')
    file.close()

    print('Building Binary Search Tree...\n')

    print('Binary Search Tree stats:\n')
    print('Number of nodes:', NodeCount(T))
    print('Height:', GetHeight(T))
    print('Running time for Binary Search Tree construction: '+ str(round((end - start), 6))+'\n')

    # Call Similarity function with T and "Find"
    Similarity(T, SearchBST, 'pairs.txt', 300)

    # Similarity test for more words
    yn = input('Test similarities again with random words? Y/N ')
    if yn.lower() == 'y':
        num_pairs=int(input('Enter number of random pairs (up to 9999): '))
        if num_pairs <= 9999:
            Similarity(T, SearchBST, 'pairs10000.txt', num_pairs)

def Similarity(T,choice, file_choice, num_pairs):

#    numpairs = int(input('Enter # of pairs to compare: ') )

    # Open and read pairs word file and insert into a list as list of pairs
    pairs=[line.strip().split(' ') for line in open(file_choice,'r')]

    # Assign timer to 0
    timer=0

    # Loop to iterate through list of pairs line by line
    for i in range(num_pairs):
        # Start timer
        start = time.perf_counter()

        # word1 gets the first column in each line from pairs list
        word1=pairs[i][0]
        # word2 gets the second column in each line from pairs list
        word2=pairs[i][1]

        #word1emb and word2emb gets the embedding that is found by
        #using SearchBST (for BST) or SearchBTree (for B-Tree)
        word1emb=(choice(T,word1))
        word2emb=(choice(T,word2))

        # Check if word1emb or word2emb is not found
        if word1emb is None or word2emb is None:
            continue

        # Formula to find cosine distance between both word embeddings
        cosine_distance = np.dot(word1emb.emb, word2emb.emb)/
(np.linalg.norm(word1emb.emb)* np.linalg.norm(word2emb.emb))
        # Stop timer for every iteration
```

```
        end = time.perf_counter()

        # Add each timed iteration of finding similaties to "timer"
        timer += end - start

        print(i+1,'Similarity ['+word1+','+word2+'] =',str(round(100*cosine_distance,5))+'%')

    print('\n\nRunning time for similarities with',len(pairs),'pairs: '+ str(round(timer,4)))

# Main
if __name__ == "__main__":

    print('Choose table implementation')
    print('Type 1 for binary search tree or 2 B-tree')
    choice=int(input('Choice: '))

    if choice==1:
        BST_Test()
    if choice==2:
        BTree_Test()
```

Academic Honesty Statement:

“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.”

-Laurence Labayen