Computer Science - Data Structures

**Lab#4**

Binary and B-Trees

horizontal line

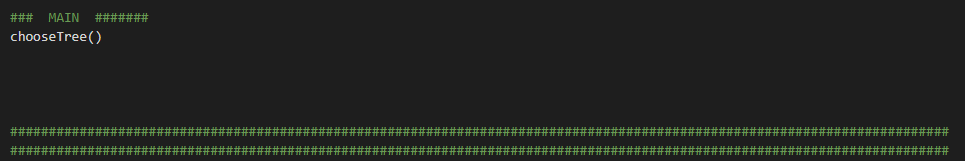
# 

# Introduction

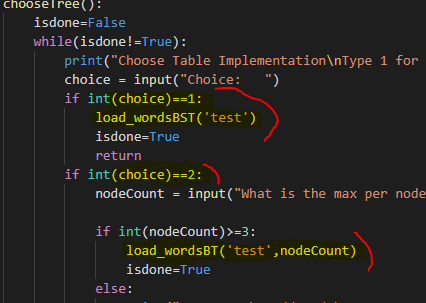
Natural Language Processing (NLP) is a subfield of artificial intelligence that deals with the design of algorithms, programs, and systems that can understand human languages in written and spoken forms.Word embeddings are a recent advance in NLP that consists of representing words by vectors (or arrays) of floating point numbers in such a way that if two words are similar, their embeddings are also similar. Seehttps://nlp.stanford.edu/projects/glove/for an overview of this interesting research.In order to work in real-time, NLP applications such as Siri and Alexa use efficient data structures to quickly retrieve the embeddings given their corresponding words. In this lab you will implement a simple version of this. In the web page mentioned above you can find links to files that contain word embeddings of various lengths for various vocabulary sizes. Use the fileglove.6B.50d.txt(included in the fileglove.6B.zip), which contains word embeddings of length 50 for a very large number of words. Each line in the file starts with the word being described, followed by 50 floating point numbers that represent the word’s vector description (the embedding). The words are ordered by frequency of usage, so ”the” is the first word. Some ”words” do not start with an alphabetic character (for example ”,” and ”.”); feel free to ignore them. If you run into memory issues, feel free to also limit the number of words you use

## Main function (in reality)

The runner area is located at the bottom of the file:



However this is not where the main code resides. Rather it falls after the choose tree function.



As seen here after the choice is made it will decide what tree to build. Though the methods are very similar, there were some differences that made it important.

### Importing and creating a tree

### 

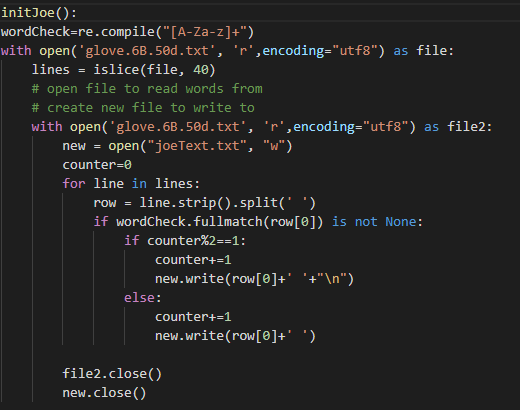
I did not read the entire file as i was testing because it took too long and even opening the txt file crashed my computer several times. So to combat this I limited the length of the file to the top N lines. I did this with the “islice” method and you will see an import at the top for this function. The initialization of the tree is roughly the same between BST and B-Tree, it checks first to see if the value is a word or something else before it adds it to the tree. If it is a word then it adds the embedded version of it to the tree.

### 

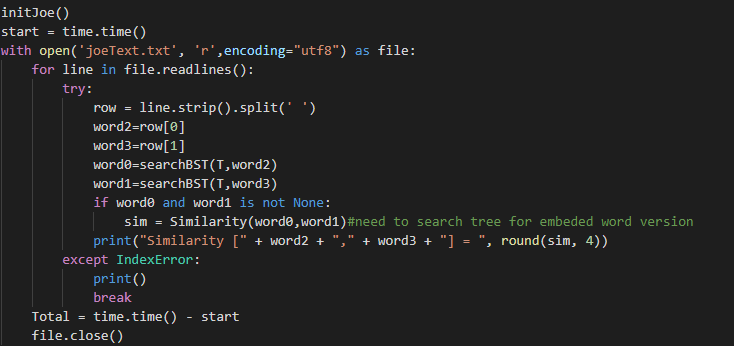
### TestFile



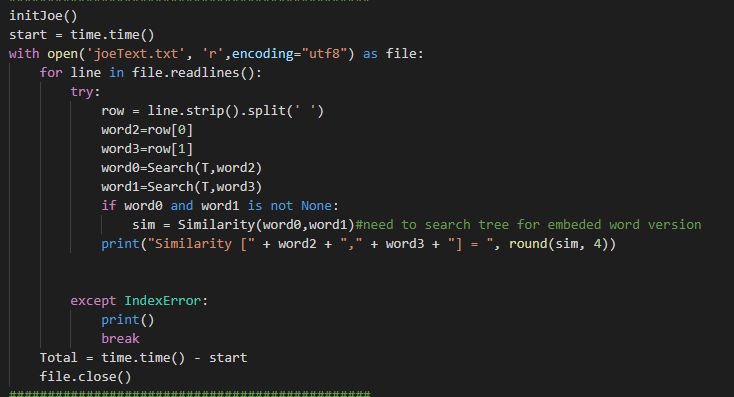
You will see this before checking similarity, all it does is open the glove file and creates a new file with 2 words per line.

 I used a %2 and a counter to determine if a line has 1 or 2 words as its creating a file.

### BST

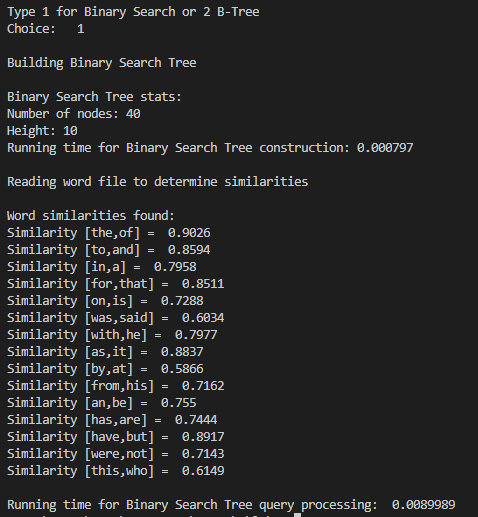
This is the core of how i compare words, i ran into an issue with just using row[0]&row[1] in the Similarity call so i stored their values in word 2&3 before calling them. This is why the word indexing is off, instead of it being 0-3 is is 2,3,0,1 as the order i made them. The SearchBST searches for a word and finds its equivalent in the tree.

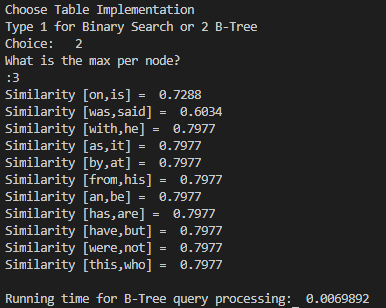
### B-Tree



Concept is the same as BST, the only differences are for operational reasons. Such as using a different search function.

### Outputs





### Appendix

'''

Thomas Roque

Lab 4

CS2302 Data Structures

MW 10:30

Professor: Olac Fuentes

TA: Anindita Nath

'''

from itertools import islice

import matplotlib.pyplot as plt

import numpy as np

import math

import time

import re

##########################################################################################################################

## Classes ###############################################################################################################

##########################################################################################################################

class BST(object):

# Constructor

def \_\_init\_\_(self, data, left=None, right=None):

self.data = data

self.left = left

self.right = right

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

class WordEmbed(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)

##########################################################################################################################

##PRE DEFINED METHODS B-Tree #############################################################################################

##########################################################################################################################

def FindChild(T,k):

# Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree

if isinstance(k, WordEmbed):

for i in range(len(T.data)):

if k.word < T.data[i].word:

return i

else:

for i in range(len(T.data)):

if k < T.data[i].word:

return i

return len(T.data)

def InsertInternal(T,i):

# T cannot be Full

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)

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):

T.data.append(i)

T.data.sort(key=lambda x: x.word)#needed to use embeded 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 InsertBT(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 Height(T):

if T.isLeaf:

return 0

return 1 + Height(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)])

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 Search(T,k):

# Returns node where k is, or None if k is not in the tree

for x in range(0, len(T.data)):

if k == T.data[x].word:

return T.data[x]

if T.isLeaf:

return None

return Search(T.child[FindChildSearch(T,k)],k)

def FindChildSearch(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 < T.data[i].word:

return i

return len(T.data)

def Set\_x(T,Dx):

# Finds x-coordinate to display each node in the tree

if T.isLeaf:

return

else:

for c in T.child:

Set\_x(c,Dx)

d = (Dx[T.child[0].data[0]] + Dx[T.child[-1].data[0]] + 10\*len(T.child[-1].data))/2

Dx[T.data[0]] = d - 10\*len(T.data)/2

def DrawBtree\_(T, Dx, y, y\_inc, fs, ax):

# Function to display b-tree to the screen

# It works fine for trees with up to about 70 data

xs = Dx[T.data[0]]

if T.isLeaf:

for itm in T.data:

ax.plot([xs,xs+10,xs+10,xs,xs],[y,y,y-10,y-10,y],linewidth=1,color='k')

ax.text(xs+5,y-5, str(itm), ha="center", va="center",fontsize=fs)

xs +=10

else:

for i in range(len(T.data)):

xc = Dx[T.child[i].data[0]] + 5\*len(T.child[i].data)

ax.plot([xs,xs+10,xs+10,xs,xs],[y,y,y-10,y-10,y],linewidth=1,color='k')

ax.text(xs+5,y-5, str(T.data[i]), ha="center", va="center",fontsize=fs)

ax.plot([xs,xc],[y-10,y-y\_inc],linewidth=1,color='k')

DrawBtree\_(T.child[i], Dx, y-y\_inc, y\_inc, fs, ax)

xs +=10

xc = Dx[T.child[-1].data[0]] + 5\*len(T.child[-1].data)

ax.plot([xs,xc],[y-10,y-y\_inc],linewidth=1,color='k')

DrawBtree\_(T.child[-1], Dx, y-y\_inc, y\_inc, fs, ax)

def DrawBtree(T):

#Find x-coordinates of leaves

LL = Leaves(T)

Dx ={}

d =0

for L in LL:

Dx[L[0]]=d

d += 10\*(len(L)+1)

#Find x-coordinates of internal nodes

Set\_x(T,Dx)

#plt.close('all')

fig, ax = plt.subplots()

DrawBtree\_(T, Dx, 0, 30, 10, ax)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

##########################################################################################################################

##PRE DEFINED METHODS BST ################################################################################################

##########################################################################################################################

def Insert(T,newItem):

if T == None:

T = BST(newItem)

elif T.data.word > newItem.word:

T.left = Insert(T.left,newItem)

else:

T.right = Insert(T.right,newItem)

return T

def InOrder(T):

if T is not None:

InOrder(T.left)

print(T.data,end=' ')

InOrder(T.right)

##########################################################################################################################

##########################################################################################################################

##########################################################################################################################

## MAIN SECTION ######################################################################################################

##########################################################################################################################

def Similarity(word0, word1):

dotProduct = np.dot(word0.emb,word1.emb)

magnitude = np.linalg.norm(word0.emb) \* np.linalg.norm(word1.emb)

return dotProduct/magnitude

#needed to spawn a file with words that i am testing

def numItems(T):

sum = len(T.data)

if not T.isLeaf:

for i in range(len(T.child)):

sum+=numItems(T.child[i])

return sum

def initJoe():

wordCheck=re.compile("[A-Za-z]+")

with open('glove.6B.50d.txt', 'r',encoding="utf8") as file:

lines = islice(file, 40)

# open file to read words from

# create new file to write to

with open('glove.6B.50d.txt', 'r',encoding="utf8") as file2:

new = open("joeText.txt", "w")

counter=0

for line in lines:

row = line.strip().split(' ')

if wordCheck.fullmatch(row[0]) is not None:

if counter%2==1:

counter+=1

new.write(row[0]+' '+"\n")

else:

counter+=1

new.write(row[0]+' ')

file2.close()

new.close()

def searchBST(T,key):

if T is None:

return

if T.data.word==key:

return T.data

if T.data.word < key:

return searchBST(T.right,key)

return searchBST(T.left,key)

def BSTHeight(T):

if T is None:

return 0

else :

leftSide = BSTHeight(T.left)

rightSide = BSTHeight(T.right)

return max(leftSide,rightSide)+1

def TreeCount(T):

sum=0

if T is not None:

sum+=1

sum= sum + TreeCount(T.left)

sum= sum + TreeCount(T.right)

return sum

def load\_wordsBST(WORDLIST\_FILENAME):

# Pattern to be used to remove words with unwanted characters

wordCheck=re.compile("[A-Za-z]+")

with open('glove.6B.50d.txt', 'r',encoding="utf8") as file:

start = time.perf\_counter()

lines = islice(file, 50)

#replace lines with file.readlines() for full file

T=None

for line in lines:

row = line.strip().split(' ')

if wordCheck.fullmatch(row[0]) is not None:

T=Insert(T, WordEmbed(row[0],[(i) for i in row[1:]]))

# Stop timer

end = time.perf\_counter()

file.close()

print('\nBuilding Binary Search Tree\n')

print('Binary Search Tree stats:')

print('Number of nodes:', TreeCount(T))

print('Height:', BSTHeight(T))

print('Running time for Binary Search Tree construction: '+ str(round((end - start), 6))+'\n')

print("Reading word file to determine similarities\n")

print("Word similarities found:")

###############################################

initJoe()

start = time.time()

with open('joeText.txt', 'r',encoding="utf8") as file:

for line in file.readlines():

try:

row = line.strip().split(' ')

word2=row[0]

word3=row[1]

word0=searchBST(T,word2)

word1=searchBST(T,word3)

if word0 and word1 is not None:

sim = Similarity(word0,word1)#need to search tree for embeded word version

print("Similarity [" + word2 + "," + word3 + "] = ", round(sim, 4))

except IndexError:

print()

break

Total = time.time() - start

file.close()

###############################################

print("Running time for Binary Search Tree query processing: ", round(Total, 7))

def load\_wordsBT(WORDLIST\_FILENAME,maxNum):

wordCheck=re.compile("[A-Za-z]+")

with open('glove.6B.50d.txt', 'r',encoding="utf8") as file:

T = BTree([], [], max\_data=int(maxNum)) # init

# Start counter

start = time.perf\_counter()

lines = islice(file, 20)#LIMITS SIZE FOR TESTING

#replace lines with file.readlines()

for line in lines:

row = line.strip().split(' ')

if wordCheck.fullmatch(row[0]) is not None:

InsertBT(T,WordEmbed(row[0],[(i) for i in row[1:]]))

# Stop counter

end = time.perf\_counter()

file.close()

print('\nBuilding B-Tree\n')

print('B-Tree stats:')

print('Number of nodes:', numItems(T))

print('Height:', Height(T))

print('Running time for B-Tree construction: '+ str(round((end - start), 6))+'\n')

print("Reading word file to determine similarities\n")

print("Word similarities found:")

###############################################

initJoe()

start = time.time()

with open('joeText.txt', 'r',encoding="utf8") as file:

for line in file.readlines():

try:

row = line.strip().split(' ')

word2=row[0]

word3=row[1]

word0=Search(T,word2)

word1=Search(T,word3)

if word0 and word1 is not None:

sim = Similarity(word0,word1)#need to search tree for embeded word version

print("Similarity [" + word2 + "," + word3 + "] = ", round(sim, 4))

except IndexError:

print()

break

Total = time.time() - start

file.close()

###############################################

print("Running time for B-Tree query processing: ", round(Total, 7))

DrawBtree(T)

#Chooses a tree and inits it with the word list

def chooseTree():

isdone=False

while(isdone!=True):

print("Choose Table Implementation\nType 1 for Binary Search or 2 B-Tree")

choice = input("Choice: ")

if int(choice)==1:

load\_wordsBST('test')

isdone=True

return

if int(choice)==2:

nodeCount = input("What is the max per node?\n:")

if int(nodeCount)>=3:

load\_wordsBT('test',nodeCount)

isdone=True

else:

print("Max must be odd and be greater than or equal to 3")

return

else:

print("Enter a number between 1 and 2 please")

### MAIN #######

chooseTree()

##########################################################################################################################

##########################################################################################################################