**Introduction**

The objective of this lab is to find similarity between two words using embedding numbers. In order to accomplish this, in this lab, there are two ways which are by binary search tree and by hash table. By getting the solution with two different ways, we can compare the run time and know the efficiency of algorithm.

**Part-1: building Binary Search Tree**

Binary search tree is a tree which has two children at max and the location of the value is fixed depends on the value of inserting. The value that I am inserting is an array containing a word (string) and an numpy array of fifty floating numbers. Since each noide will have an array, at first, I changed the class to let item be []. Each line in text file has one word followed by 50 float numbers, thus I am sending [word, 50 numbers converted to numpy float number] and location is fixed depend on the string. All the method to insert node, find height of tree, and count the number of nodes in tree are exactly same the lab3, the only change that needed is to change insert method from recognizing integer to a string (newItem[0]). In order to compare the runtime between binary search tree and hash table, I am getting the value of starting poing and ending point by time.time().

**Part-2: Find Similarity using BST**

We are finding out the similarity between two string read by a different text file than part2. After I split the line and have an array with two words, I use the “search\_word” which will find the embeedings form the specific word. Again, the method search\_word is very similar to find item from lab3. The only change I made is to make the method to compare string (k[0]) instead of integer. The return value is the embedding to compute the similarity using cosine. The way to compute the similarity is by using the formula given in the PDF. Summation of multiplication of two embeddings divided by distance of two embeddings. Same as part-1, we are comparing the runtime between binary search tree and hash table when computing for similarity, I have the value of starting and ending point to see how much time it took to compute similarity.

**Part-3: Build Hash Table**

Hash table type: chaining is a data structure which fixes the index where the element will be store depends on a certain manner. Same as the part1, first I read the file and parse it to have all the information in a single array. I have a if statement which takes away what is not string. Inserting an array which contains a string and 50 floating numbers. The way to decide the position of element is set by the length of the string and ascii number of the inputting string. The size of the hash table is 19 and eventually, it will exceed at some point. When load factor is 1, resize the hash table to 2\*len+1. When I do this, I made another hash table with bigger size and insert each value into a new hash table. A method which returns the location index is given by Prof. Fuentes as well as finding the words. To get the percentage of empty list is to divide number of empty list by the size of the hash table. Standard deviation of the length of the list is subtract every length from average and square it. Take a square root of of multiplication of previous statement with (1/length). Since we are compare the runtime between binary search tree and hash table, I am getting the value before building and after building the hash table.

**Part4: Find Similarity using Hash table**

In order to find the similarity, we need to search for the words that we are comparing to have embeddings. I used the sample code uploaded by Prof. Fuentes. I changed the return type from tree elements to an array which is the embedding. After getting the embeddings, the computation to find the similarity is the same as part-3 on above. Also, for this part, I am getting the before and after getting the result to compare.

**Output: Using Binary Search Tree**

Type 1 for binary search tree or 2 for hush table with chaining

Choice: 1

Building binary search tree.

Binary Search Tree stats:

Number of nodes: 327091

Height: 48

Running time for binary search tree construction: 83

Reading word file to determine similarities.

Similarity ['bear', 'bear'] = 1.0

Similarity ['bear', 'bear'] = 1.0

Similarity ['barley', 'shrimp'] = 0.5353

Similarity ['federer', 'baseball'] = 0.287

Similarity ['federer', 'tennis'] = 0.7168

Similarity ['harvard', 'stanford'] = 0.8466

Similarity ['harvard', 'utep'] = 0.0684

Similarity ['harvard', 'ant'] = -0.0267

Similarity ['raven', 'crow'] = 0.615

Similarity ['raven', 'whale'] = 0.3291

Similarity ['spain', 'france'] = 0.7909

Similarity ['spain', 'mexico'] = 0.7514

Similarity ['mexico', 'france'] = 0.5478

Similarity ['mexico', 'guatemala'] = 0.8114

Similarity ['computer', 'platypus'] = -0.1277

Running time for binary search tree query processing: 46

**Output: Using Hash Table**

Type 1 for binary search tree or 2 for hush table with chaining

Choice: 2

Building hush table.

Hash Table stats:

Initail table size 19

Total elements: 327091

Final table size: 327679

Load factor: 0.9982055609300565

Percentage of empty lists: 36.80187012289466

Standard deviation of the lengths of the lists: 0.9978705669480371

Running time for Hash Table construction: 75

Reading word file to determine similarities.

Similarity ['bear', 'bear'] = 1.0

Similarity ['bear', 'bear'] = 1.0

Similarity ['barley', 'shrimp'] = 0.5353

Similarity ['federer', 'baseball'] = 0.287

Similarity ['federer', 'tennis'] = 0.7168

Similarity ['harvard', 'stanford'] = 0.8466

Similarity ['harvard', 'utep'] = 0.0684

Similarity ['harvard', 'ant'] = -0.0267

Similarity ['raven', 'crow'] = 0.615

Similarity ['raven', 'whale'] = 0.3291

Similarity ['spain', 'france'] = 0.7909

Similarity ['spain', 'mexico'] = 0.7514

Similarity ['mexico', 'france'] = 0.5478

Similarity ['mexico', 'guatemala'] = 0.8114

Similarity ['computer', 'platypus'] = -0.1277

Running time for hash table query processing: 47

**Runtime Comparison**

|  |  |  |
| --- | --- | --- |
|  | Binary Search Tree | Hash Table |
| Building | 83 | 75 |
| Finding similarity | 46 | 47 |

"""""""""""""""""""""""""""""""""""""""

Lab 5 - Find Similarity

04/1/2019 - Ken M. Amamori

CS2302 MW 10:30 - 11:50

Professor: Olac Fuentes

TA: Anindita Nath, Maliheh Zargaran

"""""""""""""""""""""""""""""""""""""""

import numpy as np

import time

import math

""""""""""""

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

def \_\_init\_\_(self,size):

self.item = []

self.num\_items = 0

for i in range(size):

self.item.append([])

""""""""""""

class BST(object):

# Constructor

def \_\_init\_\_(self, item=[], left=None, right=None):

self.item = item

self.left = left

self.right = right

""""""""""""

#Build HashTable

def ht(f, f2):

print("\nBuilding hush table.\n")

print("Hash Table stats:")

H = HashTableC(19) #create Hash Table of length 17

print("Initail table size", len(H.item))

start = int(time.time()) #starting time

for line in f: #read line by line, glove

data = line.split(' ')

if data[0].isalpha():

H = InsertC(H, data) #insert data

end = int(time.time()) #ending time

print("Total elements: ", H.num\_items)

print("Final table size: ", len(H.item))

print("Load factor: ", H.num\_items/len(H.item))

c, d = infolist(H)

print("Percentage of empty lists:", c/len(H.item)\*100)

print("Standard deviation of the lengths of the lists:", d)

print(H.item[int(d)+1])

print("Running time for Hash Table construction:", (end-start))

print("\nReading word file to determine similarities.\n")

start2 = int(time.time()\*1000)

for line2 in f2: #read line by line, word\_pair

data2 = line2.split(',')

e0 = FindC(H, data2[0]) #return array if string found

e1 = FindC(H, data2[1]) #return array if string found

print("Similarity", data2[0:2], " = ", round(np.sum(e0\*e1)/(math.sqrt(np.sum(e0\*e0))\*math.sqrt(np.sum(e1\*e1))),4)) #compute the similarity

end2 = int(time.time()\*1000) #ending time

print("\nRunning time for hash table query processing: ", (end2-start2))

#HT: return # of empty list and standard deviation of lengths of lists

def infolist(H):

c=0

m = H.num\_items/len(H.item)

k=0

for a in H.item:

k += (len(a)-m)\*(len(a)-m)

if a==[]: #[] found

c+=1

return c, math.sqrt(k\*(1/(len(H.item))))

#HT:double the size of hashtable

def doubleSize(H):

H2 = HashTableC(2\*len(H.item)+1) #size = 2\*length+1

for a in H.item: #traverse table

if a!=[]: #not empty

for i in a: #traverse node since chaining

H2.item[h(i[0], len(H2.item))].append([i[0], i[1]])

H2.num\_items+=1

return H2

#HT: insert k in H

def InsertC(H,k):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

if H.num\_items//len(H.item)==1: #recize table

H = doubleSize(H)

b = h(k[0],len(H.item)) #get the right index

H.item[b].append([k[0], np.array(k[1:]).astype(np.float)])

H.num\_items+=1 #keep up with elements

return H

#HT: return the index to insert

def h(s,n):

r = 0

t=0

"""for p in s:

t += (n\*math.sin(math.radians(3.599999\*(ord(p)-97))))

return int(t//len(s))"""

for c in s:

r = (r\*60 + ord(c))% n

return r

#HT: find k and return array if found

def FindC(H,k):

# Returns bucket (b) and index (i)

# If k is not in table, i == -1

b = h(k,len(H.item)) #get index

for i in range(len(H.item[b])): #traverse the node

if H.item[b][i][0] == k: #found

return H.item[b][i][1] #return array

return -1

#Build BST

def bst(f, f2):

print("\nBuilding binary search tree.\n")

T = None

start = int(time.time()) #starting time

for line in f: #get line by line

data = line.split(' ') #array separated by ' '

if data[0].isalpha():

T = Insert(T, [data[0], np.array(data[1:]).astype(np.float)]) #insert word+embeddings

end = int(time.time()) #ending time

print("Binary Search Tree stats:")

print("Number of nodes: ", count\_nodes(T)) #num of nodes

print("Height: ", find\_height(T)) #num of height

print("Running time for binary search tree construction:", (end-start))

print("\nReading word file to determine similarities.\n")

start = int(time.time()\*1000) #starting time

for line2 in f2: #word pairs

data2 = line2.split(',') #words pair separated by ','

e0 = search\_word(T, data2[0]) #search the 1st word, return array

e1 = search\_word(T, data2[1]) #search the 2nd word, return array

print("Similarity", data2[0:2], " = ", round(np.sum(e0\*e1)/(math.sqrt(np.sum(e0\*e0))\*math.sqrt(np.sum(e1\*e1))),4)) #compute the similarity

end = int(time.time()\*1000) #ending time

print("\nRunning time for binary search tree query processing: ", (end-start))

#BST: insert newitem into T

def Insert(T, newItem):

if T == None:

T = BST(newItem)

elif T.item[0] > newItem[0]:

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

else:

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

return T

#BST: find the height of a tree

def find\_height(T):

if T is not None: #base case

return (1+max([(find\_height(T.left)), find\_height(T.right)])) #1 + (the higher number)

else:

return -1

#BST: count the number of nodes in T

def count\_nodes(T):

if T is not None:

return 1 + count\_nodes(T.left) + count\_nodes(T.right)

return 0

#BST: search a string in the tree T, return array if it was found, None otherwise

def search\_word(T, k):

temp = T #temporary variable for T

while temp is not None: #iterate through necessary nodes

if temp.item[0] == k: #found

temp.item[1]

return temp.item[1]

elif temp.item[0] > k: #smaller

temp = temp.left

else: #larger

temp = temp.right

return None #not found

c = input("Type 1 for binary search tree or 2 for hush table with chaining\nChoice: ")

f = open('glove.6b/glove.6B.50d.txt', encoding='utf-8') #file with vectors

f2 = open('word\_pair.txt', encoding='utf-8') #file with pairs

if c=='1': #binary search tree

bst(f, f2)

elif c=='2': #hash table

ht(f, f2)

f.close()

f2.close()

print()

**Conclusion**:

Throughout this lab, I learned how to manage binary ever more and hash table. For the hash table was my first time to utilize, thus, it give me a good practice and get me ready for the exam. I found that hash table is faster however, it is more difficult to inplement the methds. Once you know the structure well, it is not that hard to use it. This lab was different from the others because it was very interesting and the concept comes from a study that is going on right now. In addition, it was my first time to implement something in two way to compare and see the better solution. Therefore, in this lab, I have learned and experienced new things what I can utilize in the future.

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.