Jon Munoz

Lab 2

CS2302

Professor: Olac Fuentes

TA: Anindita Nath

**INTRODUCTION**

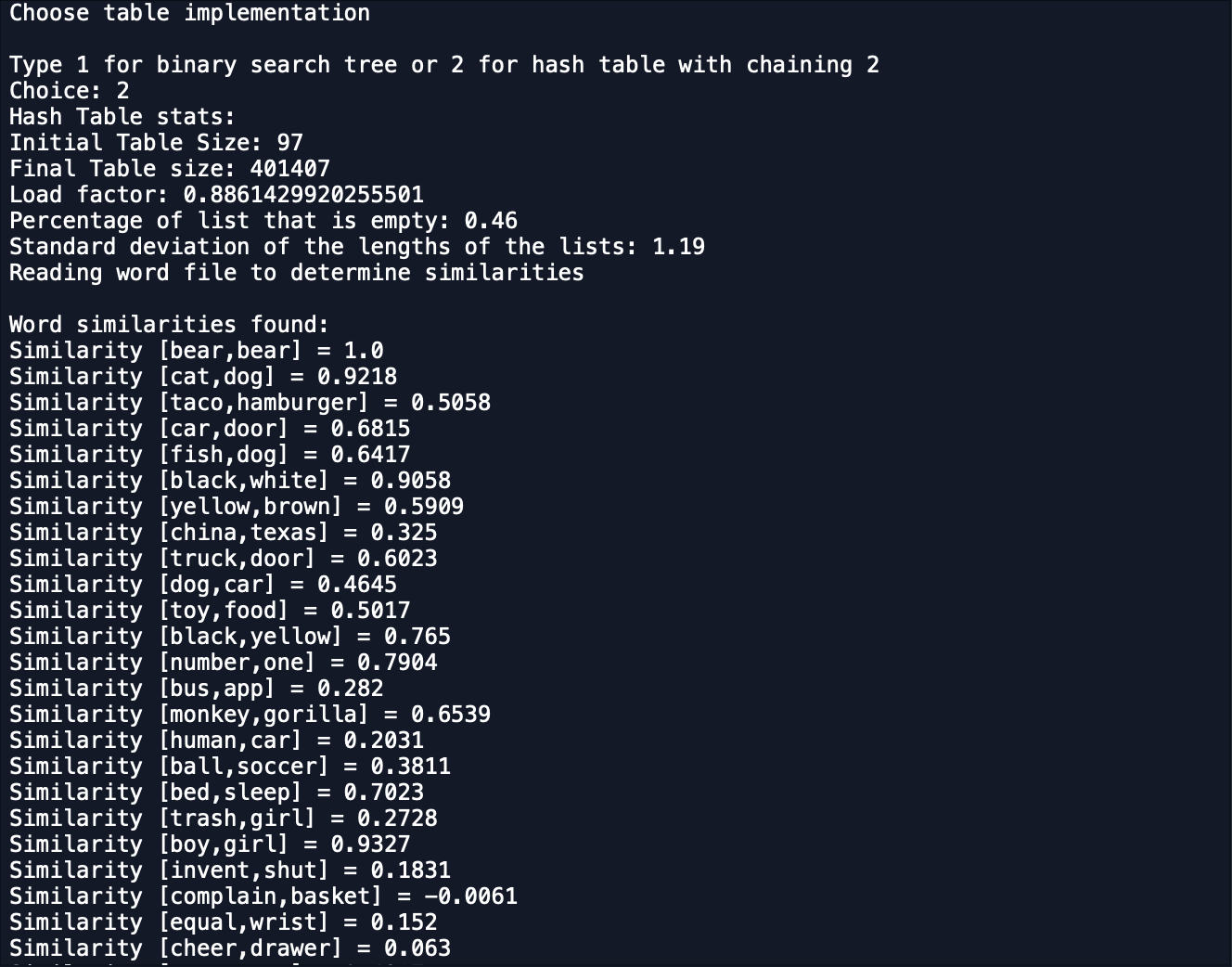
For this lab we had to work with a study conducted by Stanford regarding words and their embedding. We had to make two different algorithms to get the similarity of two worde with one way being a BST and the other being a hash table. We then had to see which one was faster in computing the similarities between a file that contained pairs of words. I personally believed that the hash table implementation would work the fastest.

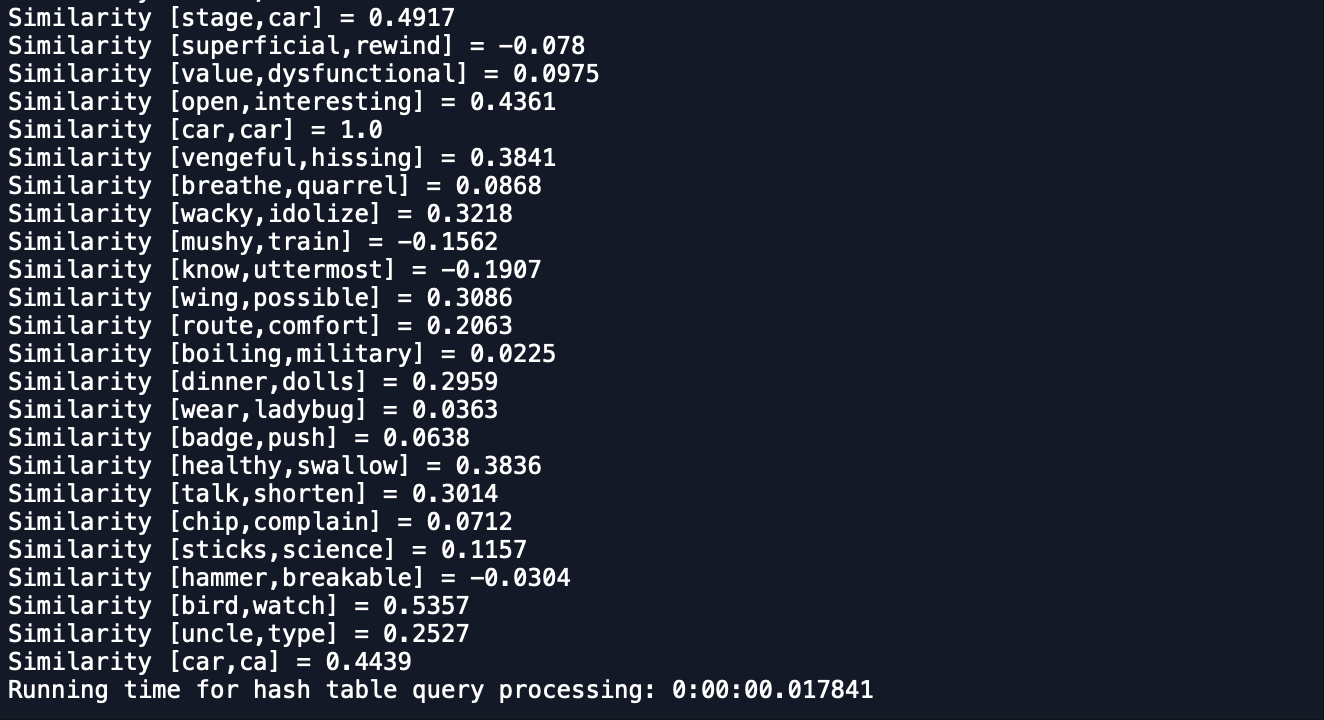
**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION**

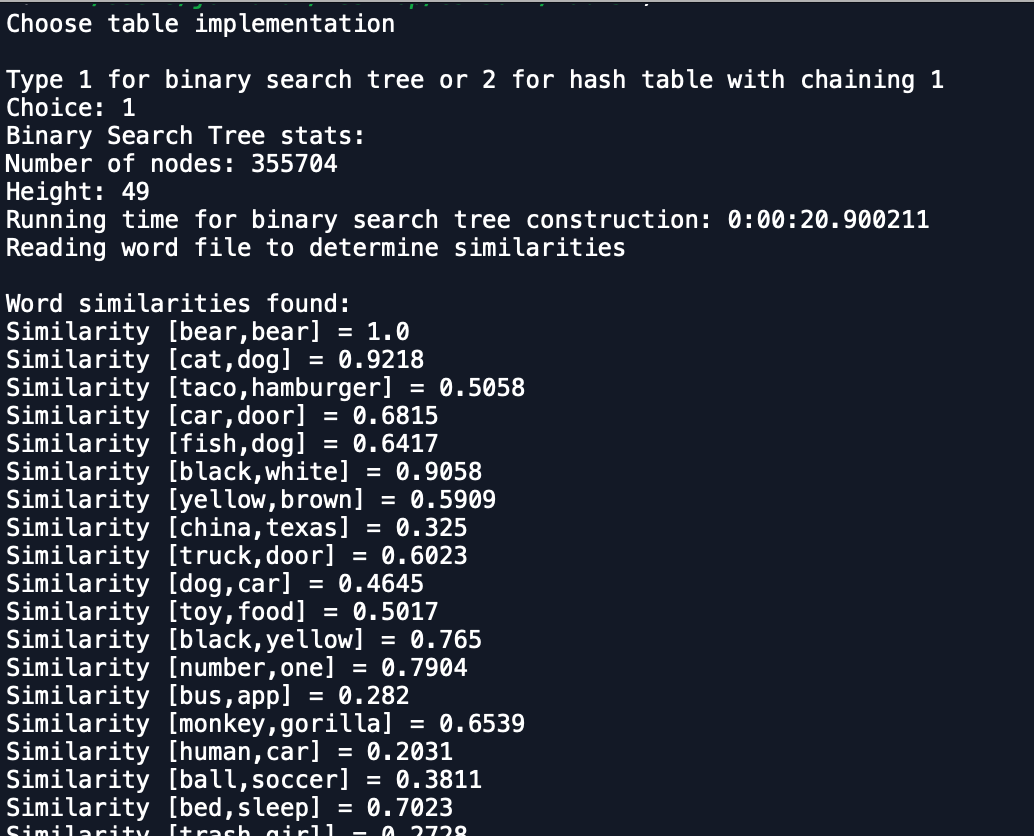
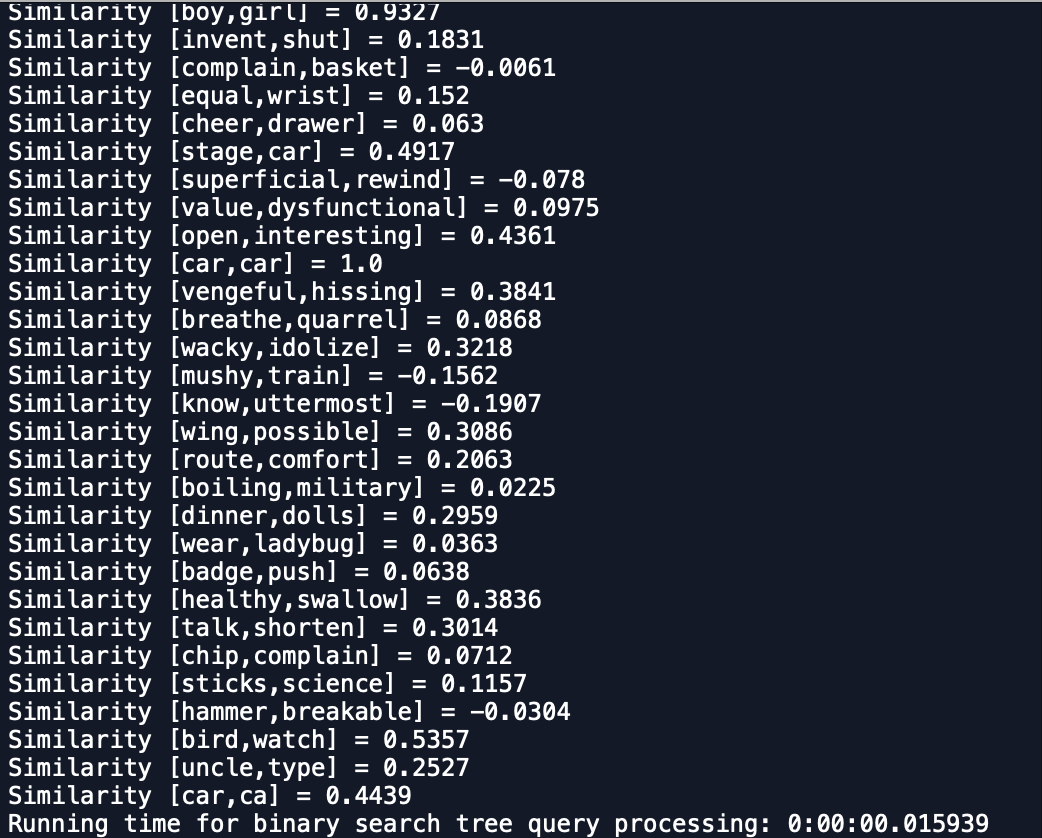
For this lab I knew I had to start off by reading the file because without reading the file I would have nothing to work with and therefore be making pointless methods. Originally when I read the file I was able to read but once I was done reading I did not know what to do at that point. I thought of putting the words into a list and the embeddings into a separate list and then from there adding them into the tree or hash table but this was redundant and would take long to do. My next approach I received help from the professor and went about it by “creating” a word and their embedding for each line and then inserting them together into the tree or table as I went along the file line by line. In order to do this I had to find the index of the first space character since that index would separate the word and its embedding. Once I found this index I separated the word and the embedding as mentioned above. This allowed me to fill the tree or table. Once I had my implementation filled I needed to create a method that would return the similarity value of two given words. This method was very similar for both implementations. What I did was make two variables, one to store the embedding of the first word and another to store the embedding of the second word, and with those two variable values I would use the equation that the professor gave us. In order to do the calculations for the equation I used two built in functions of python, those being np.dot() and np.linalg.norm(), to get the required values and then go on from there. Once I had this method I was able to move on to the query method which utilized the list created when reading the file with pairs of words. For this method the two implementations were again very similar. What I did was create a for loop that started at 0 and went all the way to the end of the list and inside this for loop I called the similarity function for each pair of words in the list. When getting the height of the BST all I did was create a for loop that went to the very bottom of the tree and had a “counter” that incremented by 1 every time that a recursive call was made and returned the final value. For the number of nodes in the tree I went through the entire tree recursively and incremented my counter by one for each recursive call. When doubling the hash table size what I did was create a new hash table in my method with the new size and I made a for loop that filled this new table with the items from the old table. For the percentage of empty list all I did was create a method that went through the entirety of the table and incremented a counter each time I found an empty bucket and returned the final number. Once I had this number I divided it by the length of the table. I also made a function that made a list of the length of each bucket that I used to get the standard deviation. This method went through the entirety of the table and appended each buckets length.

**EXPERIMENTAL RESULTS**

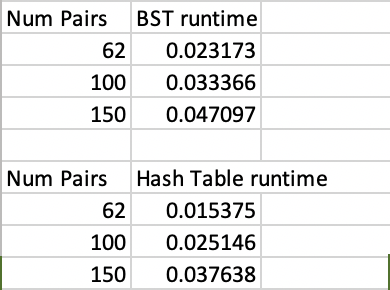
Below I have the results of calling the two different implementations, the BST and hash table. The first pictures are the hash table implementation and the second set of pictures are the BST implementation. Both runs used the same file for the pairs of words as to not have that influence the results.

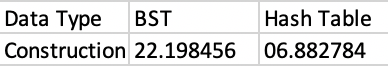
­­­­­­





Below are the runtimes of the of creating the tables or trees and then the time for the querying of each implementation and each time is in seconds(1.0 is one second, .05 is a fifth of a second):





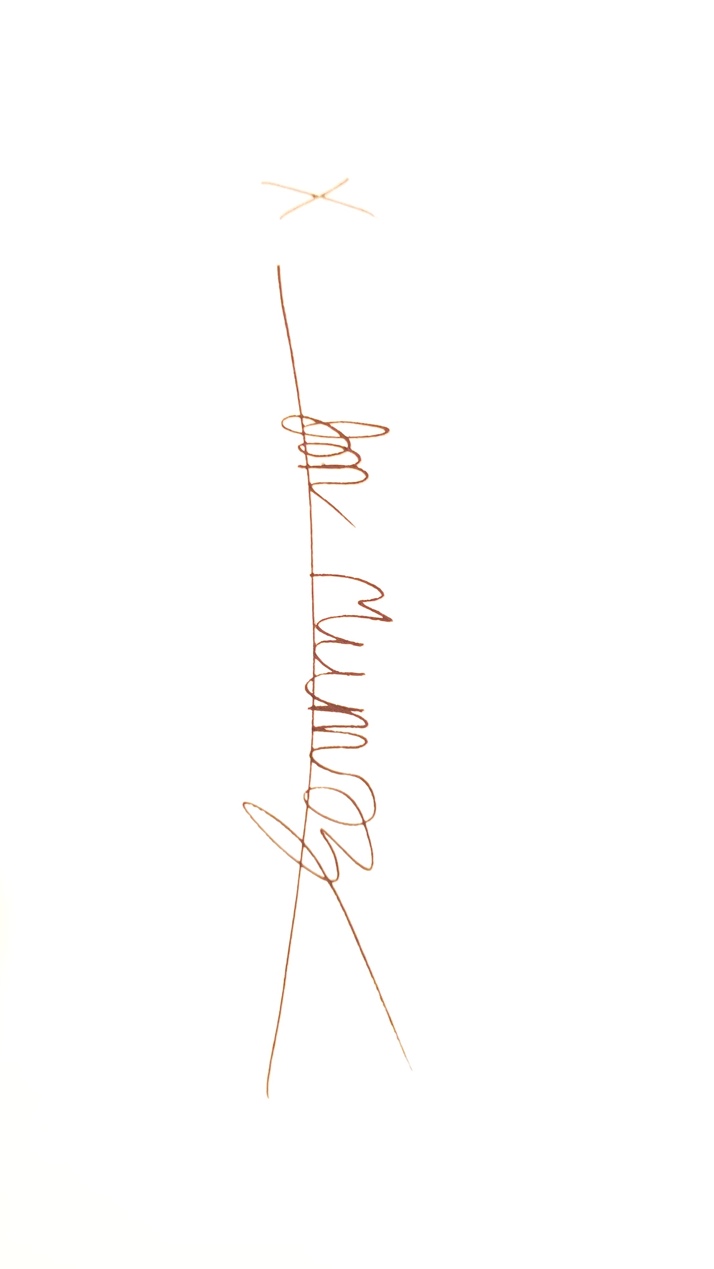
**CONCLUSION**

At the end of this lab I learned some new techniques in python such as the dot product and standard deviation. I also learned how to read files in python which was completely shocking when I saw how simple it was compared to java. One last thing I learned was that for this assignment and probably most other implementations hash tables are faster than Binary search trees.

**APPENDIX**

**SOURCE CODE**

|  |
| --- |
| #Jon Munoz |
|  | #CS2302 Data Structures |
|  | #Lab 5 |
|  | #Instructor:Olac Fuentes |
|  | #TA:Anindita Nath |
|  | #Last Modified 4/1/19 |
|  |  |
|  |  |
|  | import datetime |
|  | import numpy as np |
|  | import statistics |
|  |  |
|  | ############################################## |
|  | class HashTableC(object): |
|  | def \_\_init\_\_(self,size): |
|  | self.item = [] |
|  | for i in range(size): |
|  | self.item.append([]) |
|  |  |
|  | def InsertC(H,k,l): |
|  | b = h(k,len(H.item)) |
|  | H.item[b].append([k,l]) |
|  |  |
|  | def FindC(H,k): |
|  | # Returns bucket (b) and index (i) |
|  | # If k is not in table, i == -1 |
|  | b = h(k,len(H.item)) |
|  | for i in range(len(H.item[b])): |
|  | if H.item[b][i][0] == k: |
|  | return b, i, H.item[b][i][1] |
|  | return b, -1, -1 |
|  |  |
|  |  |
|  | def h(s,n): |
|  | r = 0 |
|  | for c in s: |
|  | r = (r\*7 + ord(c))% n |
|  | return r |
|  |  |
|  | #this method returns the number of items in the hash table |
|  | def num\_items(H): |
|  | numItems = 0#set the original counter to 0 |
|  | for i in range(len(H.item)):#for loop goes through the entire table and counts the number of items |
|  | numItems += len(H.item[i])#add to the counter |
|  | return numItems |
|  |  |
|  | #this method gets the similarity between two given words |
|  | def simH(W1,W2): |
|  | e0 = FindC(H,W1)[2]#e0 is the first words embedding that we retrieve from the find method |
|  | e1 = FindC(H, W2)[2]#e1 is the second words embedding that we retrieve from the find method |
|  | sim = (np.dot(e0, e1))/(np.linalg.norm(e0) \* np.linalg.norm(e1))#sim is the result of doing the dot product divided by the magnitude of the two words |
|  | return sim |
|  |  |
|  | #doubles the size of the given hash table |
|  | def double(H): |
|  | newTable = HashTableC(((len(H.item)) \* 2) + 1)#initialize the new size of the new table |
|  | for i in range(len(H.item)):#the following two for loops go through the hash table and fill up the new table with the old table values |
|  | for j in range(len(H.item[i])): |
|  | if H.item[i] == None:#if the current node is empty dont add anything |
|  | print() |
|  | else: |
|  | InsertC(newTable,H.item[i][j][0],H.item[i][j][1])#add the old item to new table |
|  | return newTable |
|  |  |
|  | #compares two words from a passed list of words |
|  | def queryH(H): |
|  | for i in range(len(H)):#goes through entire list |
|  | x = round(simH(H[i][0], H[i][1]), 4)#sim is the similarity value between two words |
|  | print("Similarity [" + H[i][0] + "," + H[i][1] + "] =", x) |
|  |  |
|  | #gets load factor of hash table |
|  | def loadFactor(H, items): |
|  | return items/len(H.item) |
|  |  |
|  | #this method will give me the length of each bucket and add it into a list that I use when getting the standard deviation |
|  | def lenOfList(H): |
|  | L = []#create the list that ill store the lengths in |
|  | for i in range(len(H.item)): |
|  | L.append(len(H.item[i]))#append the length of the bucket |
|  | return L |
|  |  |
|  | #this method checks and counts how many empty buckets there are in the table to use in my percent empty |
|  | def numEmpty(H): |
|  | count = 0 #counter to track empty |
|  | for i in range(len(H.item)): |
|  | if len(H.item[i]) == 0:#if the length of the bucket is 0 then its empty so increment counter |
|  | count += 1 |
|  | return count |
|  | ############################################## |
|  |  |
|  | ############################################## |
|  | class BST(object): |
|  | def \_\_init\_\_(self, item, left=None, right=None): |
|  | self.item = item |
|  | self.left = left |
|  | self.right = right |
|  |  |
|  | 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 |
|  |  |
|  | def find(T, key): |
|  | if T is None or T.item[0] == key: |
|  | return T.item[1] |
|  | if T.item[0] < key: |
|  | return find(T.right, key) |
|  | return find(T.left,key) |
|  |  |
|  | #counts the number of nodes in a given tree |
|  | def numNodes(T): |
|  | if T == None:#if T is none then you want to return 0 since there are no nodes at this point |
|  | return 0 |
|  | else: |
|  | count = 1 |
|  | if T.right != None:#if T.right is not None then you want to continue down that tree and continually add 1 until T is None |
|  | count += numNodes(T.right) |
|  | if T.left != None:#if T.left is not None then you want to continue down that tree and continually add 1 until T is None |
|  | count += numNodes(T.left) |
|  | return count#return count (the total number of nodes) |
|  |  |
|  | #gets the height of the given tree |
|  | def getHeight(T): |
|  | if T == None:#if T is none then then you want to return 0 since there is no more depth at this point |
|  | return 0 |
|  | else: |
|  | lHeight = getHeight(T.left)#lHeight gets the height of the left |
|  | rHeight = getHeight(T.right)#rHeight gets the height of the right |
|  | if lHeight > rHeight:#compare lHeight with rHeight and if lHeight is greater then you want to return lHeight plus 1 since lHeight went down further than rHeight |
|  | return lHeight + 1 |
|  | else: |
|  | return rHeight + 1#else rHeight was bigger so you return that |
|  |  |
|  | #this method gets the similarity between two given words |
|  | def simBST(W1,W2): |
|  | e0 = find(T,W1)#e0 is the first words embedding that we retrieve from the find method |
|  | e1 = find(T, W2)#e1 is the second words embedding that we retrieve from the find method |
|  | sim = (np.dot(e0, e1))/(np.linalg.norm(e0) \* np.linalg.norm(e1))#sim is the result of doing the dot product divided by the magnitude of the two words |
|  | return sim |
|  |  |
|  | #compares two words from a passed list of words |
|  | def queryBST(L): |
|  | for i in range(len(L)):#goes through the list of words |
|  | x = round(simBST(L[i][0], L[i][1]), 4)#gets the similarity if the two words |
|  | print("Similarity [" + L[i][0] + "," + L[i][1] + "] =", x) |
|  | ############################################## |
|  |  |
|  | print("Choose table implementation") |
|  | ans = input("Type 1 for binary search tree or 2 for hash table with chaining ")#this line gets the user input |
|  | print("Choice:", ans) |
|  |  |
|  | #if the answer is 1 then we want to use the BST implementation of my code |
|  | if ans == "1": |
|  | with open("glove.6B.50d.txt", "r") as f:#line reads the file and stores it into lines |
|  | lines = f.readlines() |
|  |  |
|  | T = None#initialize tree |
|  |  |
|  | start = datetime.datetime.now()#start timing construction |
|  | for line in lines:#goes line by line |
|  | if ord(line[0]) >= ord('a') and ord(line[0]) <= ord('z'):#makes sure we only use lines that start with a letter |
|  | ind = line.index(' ')#gets the index of the first space character |
|  | word = line[:ind]#creates word |
|  | emb = np.fromstring(line[ind:-1], dtype=float, sep=' ')#creates embedding |
|  | T = Insert(T,[word, emb])#insert the word and embedding into tree |
|  | end = datetime.datetime.now()#end timing of construction |
|  | elapsed = end - start#get the elapsed time |
|  |  |
|  |  |
|  | with open("word.pairs.txt", "r") as f:#reads the file that I made that has pairs of words and stores it in lines |
|  | lines = f.readlines() |
|  |  |
|  | listOfWords = list()#list to hold the words |
|  |  |
|  | for line in lines: |
|  | if ord(line[0]) >= ord('a') and ord(line[0]) <= ord('z'):#makes sure we only use lines that start with a letter |
|  | ind = line.index(' ')#gets the index of the first space character |
|  | word1 = line[:ind]#creates first word |
|  | word2 = line[ind + 1:-1]#creates second word |
|  | listOfWords.append([word1,word2])#appends the words into the list |
|  |  |
|  |  |
|  | print("Binary Search Tree stats:") |
|  | print("Number of nodes:",numNodes(T)) |
|  | print("Height:",getHeight(T)) |
|  | print("Running time for binary search tree construction:", elapsed) |
|  | print("Reading word file to determine similarities") |
|  | print() |
|  | print("Word similarities found:") |
|  | start2 = datetime.datetime.now()#start time of query |
|  | queryBST(listOfWords) |
|  | end2 = datetime.datetime.now()#end time of query |
|  | elapsed2 = end2 - start2#total time of query |
|  | print("Running time for binary search tree query processing:",elapsed2) |
|  |  |
|  | #if the answer is 2 then we want to use the Hash Table implementation of my code |
|  | elif ans == "2": |
|  | H = HashTableC(97)#initialize hash table |
|  | initSize = len(H.item) |
|  |  |
|  | with open("glove.6B.50d.txt", "r") as f:#line reads the file and stores it into lines |
|  | lines = f.readlines() |
|  |  |
|  | count = 0#counter to keep track of number of items in table |
|  | for line in lines:#goes line by line |
|  | if ord(line[0]) >= ord('a') and ord(line[0]) <= ord('z'):#makes sure we only use lines that start with a letter |
|  | ind = line.index(' ')#gets index of first space character |
|  | word = line[:ind]#creates word |
|  | emb = np.fromstring(line[ind:-1], dtype=float, sep=' ')#creates embedding |
|  | if loadFactor(H, count) == 1:#checks to make sure load factor is 1, if it is insert double size and insert |
|  | H = double(H) |
|  | InsertC(H,word,emb) |
|  | count += 1 |
|  | else:#else just insert |
|  | InsertC(H,word,emb) |
|  | count += 1 |
|  |  |
|  | with open("word.pairs.txt", "r") as f:#reads the file that I made that has pairs of words and stores it in lines |
|  | lines = f.readlines() |
|  |  |
|  | listOfWords = list()#list to hold the words |
|  |  |
|  | for line in lines: |
|  | if ord(line[0]) >= ord('a') and ord(line[0]) <= ord('z'):#makes sure we only use lines that start with a letter |
|  | ind = line.index(' ')#gets the index of the first space character |
|  | word1 = line[:ind]#creates first word |
|  | word2 = line[ind + 1:-1]#creates second word |
|  | listOfWords.append([word1,word2])#appends the two words to the list |
|  |  |
|  |  |
|  | print("Hash Table stats:") |
|  | print("Initial Table Size:",initSize) |
|  | print("Final Table size:",len(H.item)) |
|  | print("Load factor:",loadFactor(H,num\_items(H))) |
|  | print("Percentage of list that is empty:",round((numEmpty(H)/len(H.item)),2)) |
|  | print("Standard deviation of the lengths of the lists:",round(statistics.stdev(lenOfList(H)),2)) |
|  | print("Reading word file to determine similarities") |
|  | print() |
|  | print("Word similarities found:") |
|  | start = datetime.datetime.now()#start time of query |
|  | queryH(listOfWords) |
|  | end = datetime.datetime.now()#end time of query |
|  | elapsed = end - start#total time |
|  | print("Running time for hash table query processing:",elapsed) |
|  | else: |
|  | print("Invalid input") |



“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 provide inappropriate assistance to any student in the class.”