Lab 5: Word Similarity

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

Finding the similarity between two words requires that we find the cosine distance between the two words' embedding. Each word has an embedding of 50 float numbers and in order to find the similarity, we need to find the dot product of both embeddings and divide that by the product of both their magnitudes. What we will do in this lab is explore two implementations, a hash table and a BST, and compare their running times when building the table and finding the similarities.

2. Proposed Solution Design and Implementation

The first thing that I decided to tackle was the building table, which included the reading of the file and the inserting of the items. To build the BST, I used a method called *NewTree*. *NewTree* opens the file that contains the words and their embeddings to be read. A variable, T, is also set to *None*, as it will be used as the BST. I also have a counter to keep track of the number of words in the tree. I then have a *for* loop that goes through each line of the file, and for each line, splits the contents. After that, I check if the "word" begins with an alphabetical character. If it does, I create a numpy array of size 50, fill it with the embedding, and then store the word into a variable. After that, I insert the word and the numpy array as a list of size two into the tree. If the word does not begin with an alphabetical character, it does nothing. This method returns the tree and the number of nodes.

When inserting the item into the tree, it calls the *Insert* method which checks if the current node's word is alphabetically less than or greater than the word being inserted. This happens recursively until it is placed in the correct position.

When building the Hash Table, I use a method called *NewTable* that accepts a table size as a parameter. The method first opens the file that contains the words and their embeddings to be read. It also creates an empty hash table of the given size. I then have a *for* loop that goes through each line in the file, and for each line, checks if the number of items in the table is equal to the table size. If it is, it calls and returns the *NewTable* method with a size of *(size*2)+1*. If it is not at a load factor of one, it checks if the word begins with an alphabetical character. If it does, I create a numpy array of size 50, fill it with the embedding, and then store the word into a variable. After that, I insert the word and the numpy array as a list of size two into the hash table. If the word does not begin with an alphabetical character, it does nothing. This method returns the final hash table and the size of the final table.

When inserting the items into the hash table, the *InsertC* method is called and in the *InsertC* method, the hashing method *h* is called. This method finds and returns the bucket that the item is to be placed in. Then the item is appended to its bucket.

I then decided to tackle the comparing of the two words. To compare two words in a BST, I used a method called *TreeSimilarity* that accepts a tree as a parameter. This method opens a file that contains two words on each line. For each line in the file, the line is split and both words are stored into their own variables. I then use a method called *FindEmbeddingT*, which returns the embedding of the given word. The dot product and the magnitude of the embeddings is calculated and given this, we can find the similarity of two words. This method returns the time it takes to find the similarities, excluding the time it takes to prints the similarities.

To compare two words in a hash table, I used a method called *HashSimilarity* that accepts a hash table as a parameter. This method opens a file that contains two words on each line. For each line in the file, the line is split and both words are

stored into their own variables. I then use a method called *FindEmbeddingC*, which returns the embedding of the given word. The dot product and the magnitude of the embeddings is calculated and given this, we can find the similarity of two words. This method returns the time it takes to find the similarities, excluding the time it takes to prints the similarities.

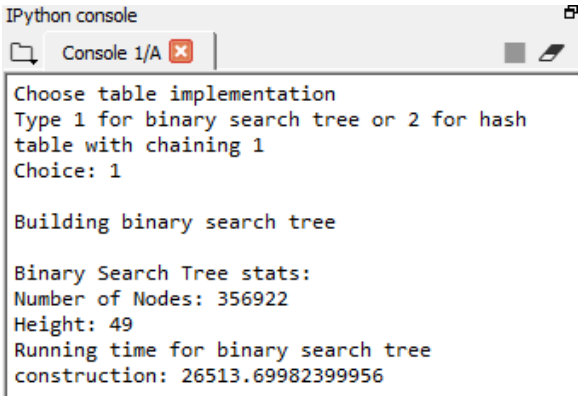
I also included other methods like *EmptyLists* that accepts a hash table as a parameter and returns the checks if a bucket contains an empty list. If it does, the counter is incremented and the method returns the ratio of empty lists.

Another method I included is called *StdDevC* that accepts a hash table as a parameter. This method returns the standard deviation of the lengths of the lists.

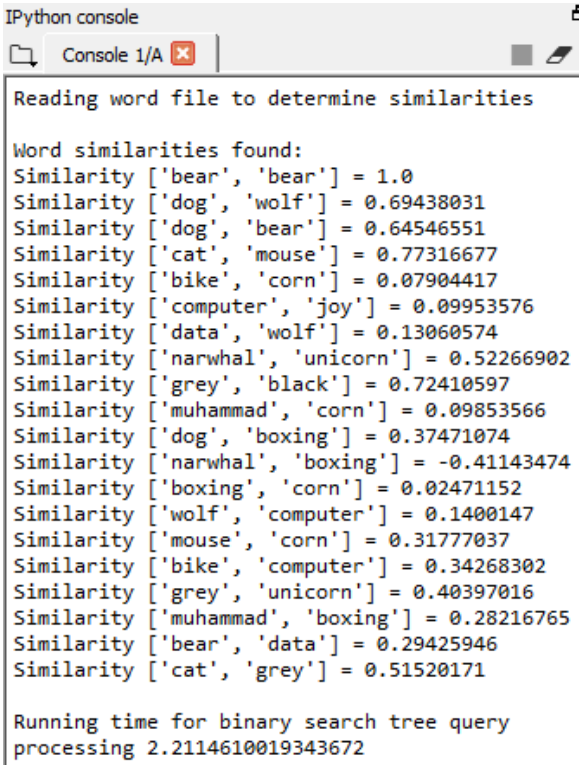
One more method I included is called *TreeHeight* and it returns the height of a given tree.

3. Experimental Results

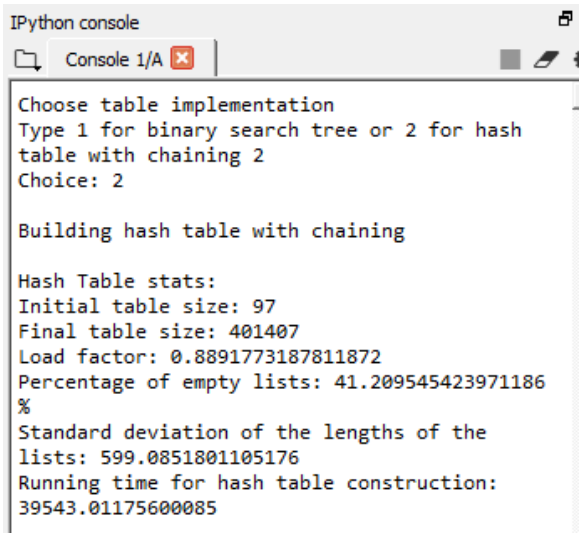
To do my experiment, I prompt the user to pick an implementation they want to use. I ran tests with both implementations using the same files *glove.6B.50d.txt* and *wordcomparison.txt*, which contained 20 word pairs. I first selected the BST. The BST was built and the tree stats were displayed, including the running time for building the tree, which was 26513.69982399956 ms.



Then the word similarites were calculated and displayed along with the running time for finding the similarities, which was 2.2114610019343672 ms.



I then selected the hash table with the same files. The hash table was built and the stats were displayed, including the running time for building the hash table, which was 39543.01175600085 ms.



Then the word similarites were calculated and displayed along with the running time for finding the similarities, which was 1.1514839970914181 ms.

```
IPython console
Console 1/A
Reading word file to determine similarities

Word similarities found:
Similarity ['bear', 'bear'] = 1.0
Similarity ['dog', 'wolf'] = 0.69438031
Similarity ['dog', 'bear'] = 0.64546551
Similarity ['cat', 'mouse'] = 0.77316677
Similarity ['bike', 'corn'] = 0.07904417
Similarity ['computer', 'joy'] = 0.09953576
Similarity ['data', 'wolf'] = 0.13060574
Similarity ['narwhal', 'unicorn'] = 0.52266902
Similarity ['grey', 'black'] = 0.72410597
Similarity ['muhammad', 'corn'] = 0.09853566
Similarity ['dog', 'boxing'] = 0.37471074
Similarity ['narwhal', 'boxing'] = -0.41143474
Similarity ['boxing', 'corn'] = 0.02471152
Similarity ['wolf', 'computer'] = 0.1400147
Similarity ['mouse', 'corn'] = 0.31777037
Similarity ['bike', 'computer'] = 0.34268302
Similarity ['grey', 'unicorn'] = 0.40397016
Similarity ['muhammad', 'boxing'] = 0.28216765
Similarity ['bear', 'data'] = 0.29425946
Similarity ['cat', 'grey'] = 0.51520171

Running time for hash table query processing
1.1514839970914181
```

The results show that the hash table had a faster running time when comparing the words than the BST.

4. Analysis of Data

When looking at the data we received from the experimental results, it becomes apparent that the hash table took a longer time to be built. This could be because the items had to be reinserted into the new hash table when it was resized. Nevertheless, the hash table was more efficient when finding the calculating the similarity between the words. Word embeddings are a good way of representing words when they need to be compared because finding the distance between the vectors yields reliable results.

5. Conclusion

With this project, I have learned how to find the similarity of two words using a BST and hash tables. At the same time, we were able to see that a hash table can be more efficient when finding the similarity between two words while it can take a little longer to build it.

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.

6. Appendix

```
'''
Course: CS2302 MW 1:30-2:50
Author: Manuel A. Ruvalcaba
Assignment: Lab #5 Hash Tables
Instructor: Dr. Olac Fuentes
TA: Anindita Nath, Maliheh Zargaran
Date of Last Modification: April 1, 2019
Purpose of the Program: The purpose of this program is to build a BST and a
                        Hash Table of words and their embeddings and find the
similarity
                        between two words, comparing the running time of both
                        data structures.
'''

class BST(object):
    # Constructor
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

class HashTableC(object):
    # Builds a hash table of size 'size'
    # Item is a list of (initially empty) lists
    # Constructor
    def __init__(self,size,num_items = 0):
        self.item = []
        for i in range(size):
            self.item.append([])
        self.num_items = num_items

import numpy as np
```

```
import math
import timeit
def InsertC(H,k,l):
    # Inserts k in appropriate bucket (list)
    # Does nothing if k is already in the table
    b = h(k[0],len(H.item))
    H.item[b].append(k)

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):
    #Returns a hash value for a given word and size of hash table
    r = 0
    for c in s:
        r = (r*53 + ord(c))% n
    return r

def NewTable(s):
    #creates a new hash table and splits each line in the file
    file = open('glove.6B.50d.txt', 'r', encoding="utf-8")
    H = HashTableC(s)
    for line in file:
        if H.num_items == s: #calls the function with the larger table size and
returns it
            file.close()
            return NewTable((s*2)+1)
```

```
        x = line.split()
        if x[0][0].isalpha():
            a = np.zeros((50,), dtype=float)
            b = x[0]
            for i in range(1,51):
                a[i-1] = x[i]
            List = [b,a]
            InsertC(H,List,len(List[0]))
            H.num_items += 1
    file.close()
    return H,s


def EmptyLists(H):
    #Finds the ratio of empty lists in the hash table
    count = 0
    for i in range(len(H.item)):
        if H.item[i] == []:
            count += 1
    return count/len(H.item)


def HashSimilarity(H):
    #finds the similarity between two words in a hash table and returns the running
time
    file = open('wordcomparison.txt','r',encoding = "utf-8")
    time = 0.0
    for line in file:
        startSim = timeit.default_timer()
        x = line.split()
        w1 = x[0]
        w2 = x[1]
        e1 = FindEmbeddingC(H,w1)
        e2 = FindEmbeddingC(H,w2)
        dotproduct = np.sum(e1*e2,dtype=float)
```

```
        magnitude =
(math.sqrt(np.sum(e1*e1,dtype=float)))*(math.sqrt(np.sum(e2*e2,dtype=float)))
        stopSim = timeit.default_timer()
        time += (stopSim-startSim)*1000
        print('Similarity',x,'=',round(dotproduct/magnitude,8))
    return time



def FindEmbeddingC(H,w):
    #finds the embedding of a word in a hash table
    b = h(w,len(H.item))
    for i in range(len(H.item[b])):
        if H.item[b][i][0] == w:
            return H.item[b][i][1]

def StdDevC(H):
    #returns the standard deviation of the length of lists
    variance = 0
    loadfactor = H.num_items/len(H.item)
    for i in range(len(H.item)):
        variance+=math.pow((len(H.item[i])-loadfactor),2)
    stdDev = math.sqrt(variance)
    return stdDev

def TreeSimilarity(T):
    #finds the similarity between two words in a BST and returns the running time
    file = open('wordcomparison.txt','r',encoding = "utf-8")
    time = 0.0
    for line in file:
        startSim = timeit.default_timer()
        x = line.split()
        w1 = x[0]
        w2 = x[1]
```

```
        e1 = FindEmbeddingT(T,w1).item[1]
        e2 = FindEmbeddingT(T,w2).item[1]
        dotproduct = np.sum(e1*e2,dtype=float)
        magnitude =
(math.sqrt(np.sum(e1*e1,dtype=float)))*(math.sqrt(np.sum(e2*e2,dtype=float)))
        stopSim = timeit.default_timer()
        time += (stopSim-startSim)*1000
        print('Similarity',x,'=',round(dotproduct/magnitude,8))
    return time


def FindEmbeddingT(T,w):
    #returns the embedding of a word in a BST
    if T is None or T.item[0] == w:
        return T
    if T.item[0]<w:
        return FindEmbeddingT(T.right,w)
    return FindEmbeddingT(T.left,w)

def Insert(T,newItem):
    #inserts a word and its embedding into a BST
    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 TreeHeight(T):
    #Gets the height of the Tree
    if T is None:
        return 0
```

```
    else:
        lheight = TreeHeight(T.left)
        rheight = TreeHeight(T.right)
        if lheight > rheight:
            return lheight+1
        else:
            return rheight+1


def NewTree():
    #creates a new BST and splits the contents of the file for inserting, returns a
running time
    file = open('glove.6B.50d.txt', 'r', encoding="utf-8")
    T = None
    count = 0
    for line in file:
        x = line.split()
        if x[0][0].isalpha():
            a = np.zeros((50,), dtype=float)
            b = x[0]
            for i in range(1,51):
                a[i-1] = x[i]
            T = Insert(T,[b,a])
            count += 1
    file.close()
    return T,count


#This is where the code is tested


choice = input("Choose table implementation\nType 1 for binary search tree or 2 for
hash table with chaining ")
print('Choice: '+choice)
print()
if choice == "1":
    print('Building binary search tree')
```

```
    print()
    start = timeit.default_timer()
    T,count = NewTree()
    stop = timeit.default_timer()
    print('Binary Search Tree stats:')
    print('Number of Nodes:',count)
    print('Height:',TreeHeight(T))
    print('Running time for binary search tree construction:', (stop - start)*1000)
    print()
    print('Reading word file to determine similarities')
    print()
    print('Word similarities found:')
    timeTree = TreeSimilarity(T)
    print()
    print('Running time for binary search tree query processing', timeTree)

elif choice == "2":
    print('Building hash table with chaining')
    print()
    start = timeit.default_timer()
    L,finalsize = NewTable(97)
    stop = timeit.default_timer()
    loadfactor = L.num_items/finalsize
    print('Hash Table stats:')
    print('Initial table size: 97')
    print('Final table size:',finalsize)
    print('Load factor:',loadfactor)
    print('Percentage of empty lists:',EmptyLists(L)*100,'%')
    print('Standard deviation of the lengths of the lists:',StdDevC(L))
    print('Running time for hash table construction:', (stop - start)*1000)
    print()
    print('Reading word file to determine similarities')
    print()
```

```
    print('Word similarities found:')
    timeHash = HashSimilarity(L)
    print()
    print('Running time for hash table query processing', timeHash)

else:
    print("Choice is invalid.")
```