**Lab Report 4: Binary Search Trees and B-Trees**

**I. Introduction**

This lab consisted on storing words and its embeddings, read from a file, on B-Trees and Binary Search Trees. After the words were stored, another file was read that contained pairs of words to be compared with their embeddings and returned how similar they were. There had to be methods to read the file into the respective tree that the user chose and another to read the word file and return the similarities. The classes for the B-Tree and Binary Search Trees had to handle receiving the object WordEmbedding that in itself contained the word and a numpy array of the embedding.

**II. Proposed solution design and implementation**

The first step was creating a the user interface in which it could choose the way that the words were going to be stored: B-Tree or BST.

After this was completed, my next step was to create a function that would read the file into the BST. This function read each line and stored the information on a WordEmbedding object, then on the tree as it read. The BST class was modified to be able to receive a WordEmbedding and sort the words alphabetically. Thankfully, python makes it very easy to sort alphabetically.

The next step was to make the function that would read into a BTree. This function was almost identical save to the call to insert the node with WordEmbedding that was for the htlp. The BTree class was also modified to handle sorting alphabetically. This was made by calling T.data.word which is the attribute of WordEmbedding that stores the word instead of just using T.data.

After these functions were successful, counting the time that a reading and storing took, counting the nodes on each tree and getting the height were added simply. Counting the nodes and getting the height were functions written on the corresponding class of the tree. For BST, counting the nodes was solved adding the current node and the node from the left and the right with recursion. Getting the height was solved getting the height from the left and right and returning the max. For BTrees, counting the nodes involved a counter, a for loop to reach each child, and recursion to keep counting until the end. Getting the height was solved with a counter and following the first child with recursion until there were no more children.

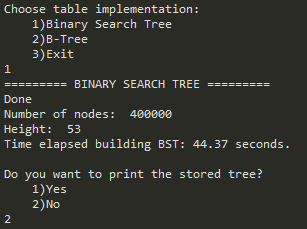
To make sure the trees were stored correctly, an optional message is sent to print the tree. The BST prints with in-order which should return the words alphabetically and the BTree prints with the structure of the BTree.

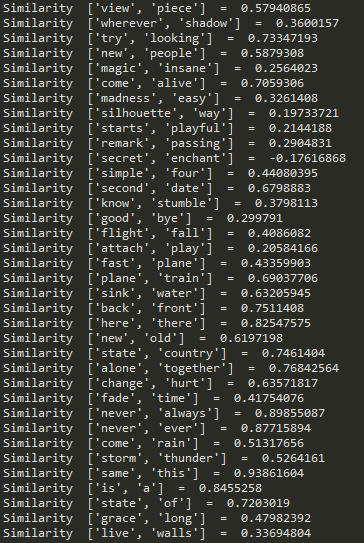
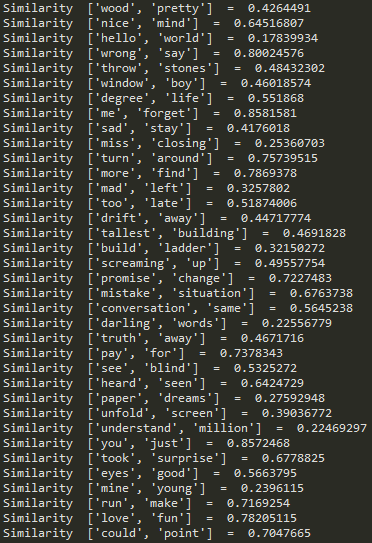
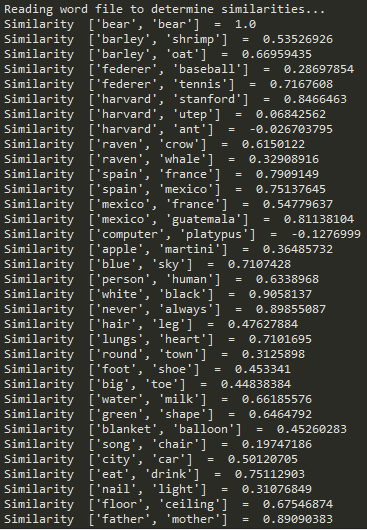
The last part was finding the similarities. This was solved with a similar function to each tree which reads the words file and looks for the word on the given tree. The call to return the embedding of the word was with a function called embedding that looked for the word in the tree and returned the embedding numpy array. After each embedding was returned for the pair of words, the formula given was applied to get the result of similarity. Time was also compared here and printed after the results were shown.

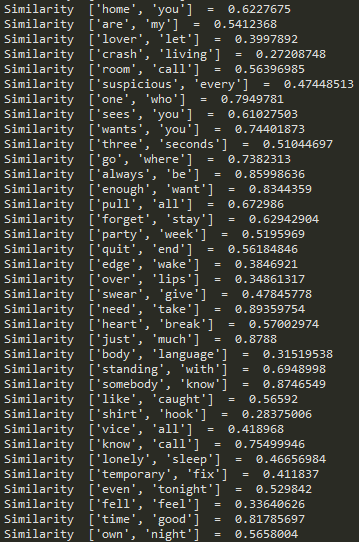
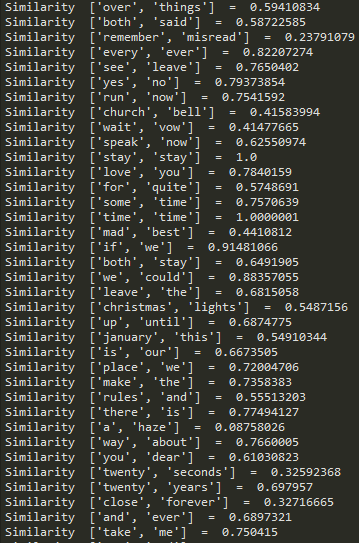
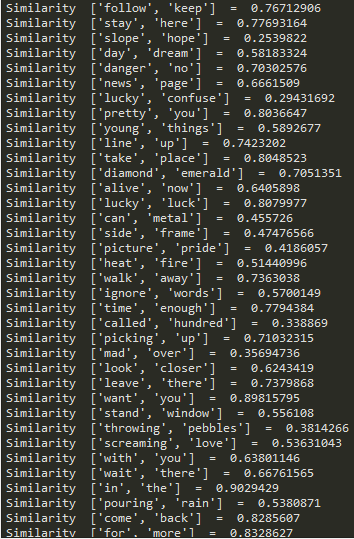
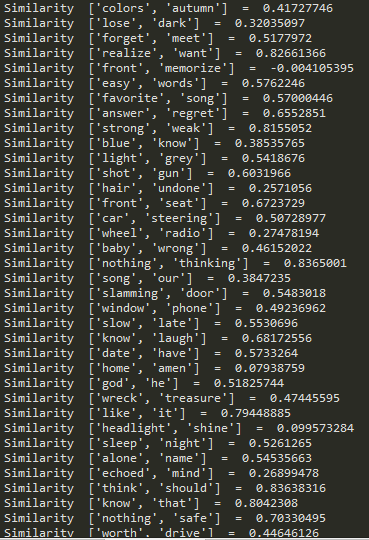
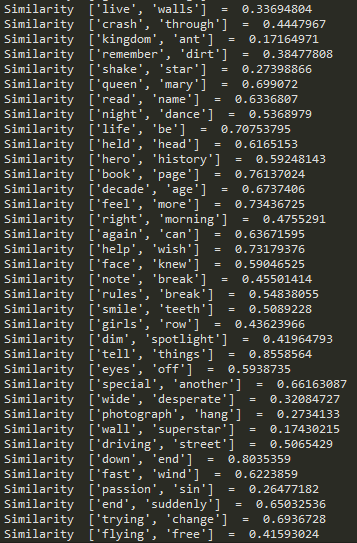
The program loops if the user wants to repeat any step or try the other tree function.

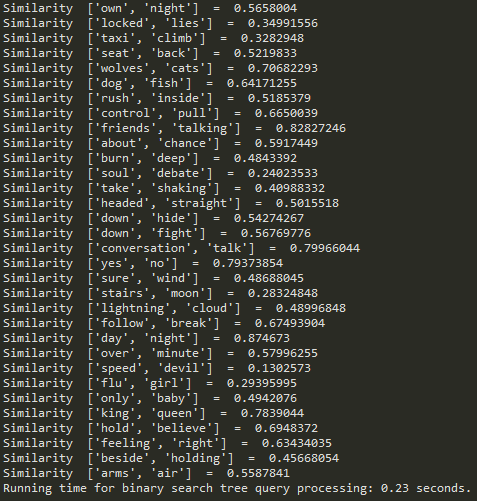
**III. Experimental results**

Testing BST:

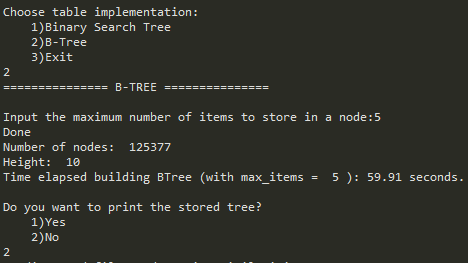
 44.37 sec

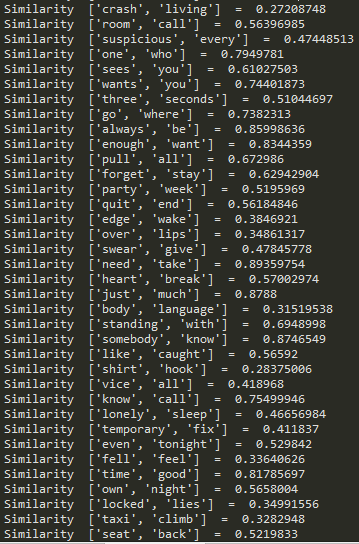
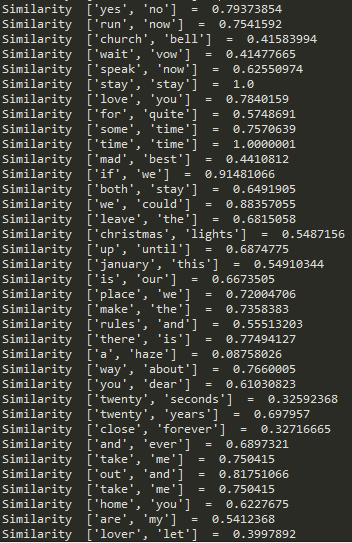
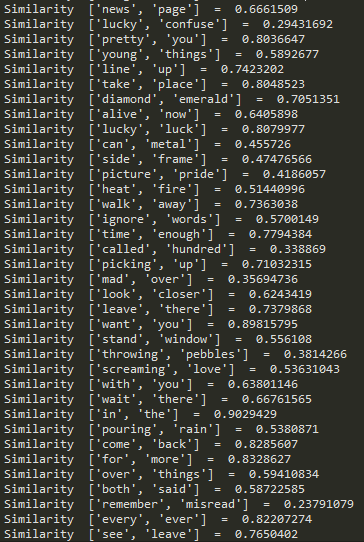
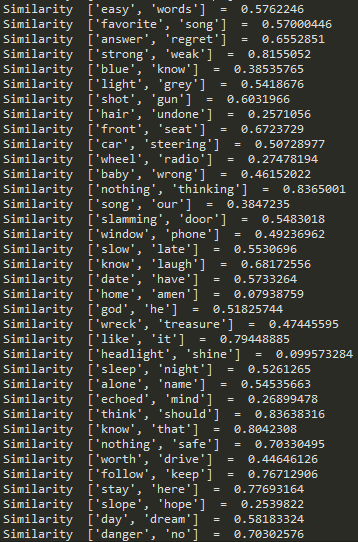
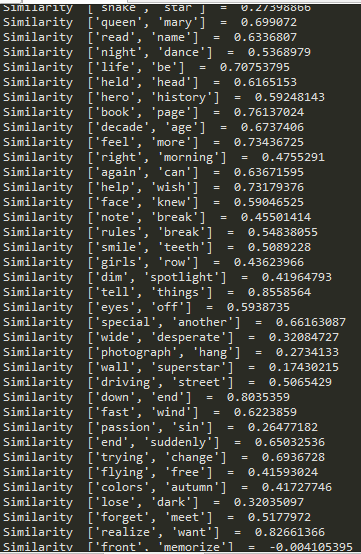
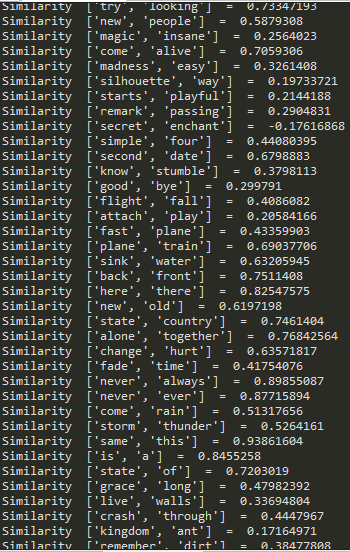
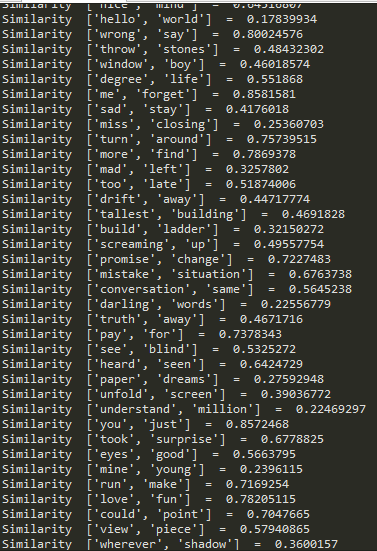
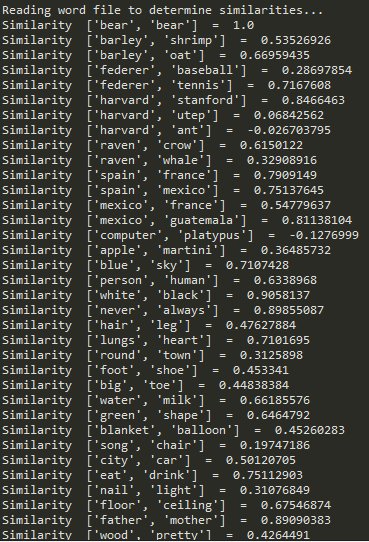


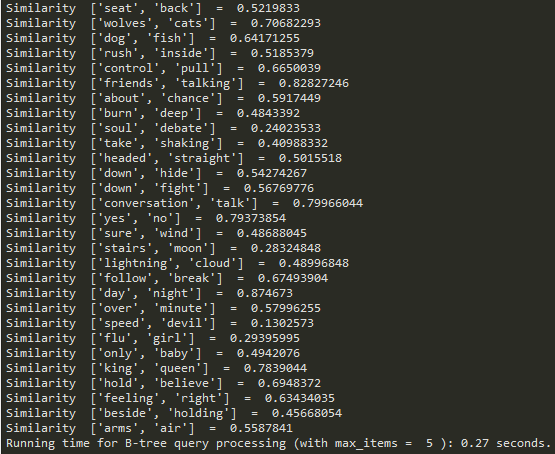


 **0.23 sec**

**Testing B-Tree**

 **59.91 sec**



 **0.27 sec**

Limiting the sizes of the input word files, I obtained the following results**:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Function** | **Reading into Tree (50,000)** | **Reading into Tree (150,000)** | **Reading into Tree (400,000)** | **Similarities (300)** | **Similarities (1000)** | **Similarities**  **(5000)** |
| Binary Search Tree | 6.07 | 16.64 | 44.37 | 0.23 | 1.05 | 5.52 |
| B-Tree(5) | 7.35 | 21.63 | 59.91 | 0.27 | 1.09 | 5.89 |
| B-Tree(20) | 11.34 | 29.86 | 85.86 | 0.33 | 1.14 | 5.79 |

**Table:**

**IV. Conclusions**

I learned that for every different structure it is important to handle the input and consider the data type. Data structures can be used with any data type but must be adapted to use the correct comparisons and correct instructions that make it work efficiently. For example: integers can be stored comparing by smaller and larger and words alphabetically.

I also practiced analyzing the time and was able to observe the time complexity of these structures. I can guess that B-Trees took longer because more comparisons, moves, and updates must be made to the overall tree than with the BST. The height of the B-Tree, though, was smaller than the BST. Also, the greater the maximum items to store in a node on a B-Tree made both processes slower.

Word Embeddings as a way of representing words seems like something unique that is very useful for these types of comparisons. Words by themselves could only be compared by the characters and length that they’d share, but adding these numbers allows words to be related differently.

**V. Appendix – Source code**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 4 - Binary Search Trees and BTrees

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 10/25/2019

Program: --------------------

"""

import bst

import btree

import WordEmbedding

import numpy as np

import time

#files to read

embed\_file = "glove.6B.50d.txt" #embeddings file

word\_file = "words.txt" #words with similarities file

#reading into a binary search tree

def readIntoBST():

tree = None

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

while content:

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on bst

tree = bst.Insert(tree, n)

content = ef.readline()

print("Done")

return tree

#reading into a BTree

def readIntoBTree(max):

tree = btree.BTree([],max\_data=max)

with open(embed\_file, encoding="utf8") as ef:

content = ef.readline()

#goes through every line on the file

while content:

word = content.split()

#creates Word Embedding with information from file

n = WordEmbedding.WordEmbedding(word[0], word[1:])

#inserts on btree

btree.Insert(tree, n)

btree.Print

content = ef.readline()

print("Done")

return tree

#prints bst in ascending order (inorder)

def printTree(T):

if T.left != None:

printTree(T.left)

print(T.data.word, end = ' ')

if T.right != None:

printTree(T.right)

#finds similarities between words stored in a BST

def similaritiesBST(T):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

#separates words

word1 = bst.Embedding(T,content[0])

word2 = bst.Embedding(T,content[1])

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#Prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

#finds similarities between words stored in a BTree

def similaritiesBTree(T):

with open(word\_file) as wf:

content = wf.readline().split()

#reads every line in the file

while content:

word1 = btree.Embedding(T,content[0])

word2 = btree.Embedding(T,content[1])

#returns similarities with similarities function

sim = (np.dot(word1,word2))/(np.linalg.norm(word1)\*np.linalg.norm(word2))

#prints values found

print("Similarity ", content, " = ", sim)

content = wf.readline().split()

return

stay = True

#looping until exit

while(stay):

option = input("Choose table implementation:\n\t1)Binary Search Tree\n\t2)B-Tree\n\t3)Exit\n")

#BST option

if (option == "1"):

print("========= BINARY SEARCH TREE =========")

#call to method and time

start = time.time()

tree = readIntoBST()

end = time.time()

#results

print("Number of nodes: ", bst.countNodes(tree))

print("Height: ", bst.Height(tree))

print("Time elapsed building BST: "+"{:.2f}".format(end-start) + " seconds.")

#display tree option

if(input("Do you want to print the stored tree?\n\t1)Yes\n\t2)No\n") == "1"):

printTree(tree)

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesBST(tree)

end = time.time()

print("Running time for binary search tree query processing: "+"{:.2f}".format(end-start) + " seconds.")

#BTree option

elif (option == "2"):

print("=============== B-TREE ===============")

max = int(input("Input the maximum number of items to store in a node:"))

#call to method and time

start = time.time()

tree = readIntoBTree(max)

end = time.time()

#results

print("Number of nodes: ", btree.countNodes(tree))

print("Height: ", btree.Height(tree))

print("Time elapsed building BTree (with max\_items = ", max, "): "+"{:.2f}".format(end-start) + " seconds.")

#display tree option

if(input("Do you want to print the stored tree?\n\t1)Yes\n\t2)No\n") == "1"):

btree.PrintD(tree, "")

#similarities

print("Reading word file to determine similarities...")

start = time.time()

similaritiesBTree(tree)

end = time.time()

print("Running time for B-tree query processing (with max\_items = ", max, "): "+"{:.2f}".format(end-start) + " seconds.")

#Exit

elif (option == "3"):

print("Thank you for using this program! Goodbye!")

stay = False

else:

print("Choose 1, 2 or 3.")

**------Binary Search Tree------**

class BST(object):

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

self.data = data

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

else:

if T.data.word > newItem.word: #alphabetical order

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

else:

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

return T

def countNodes(T):

#counts nodes in bst

if T == None:

return 0

n = 1 + countNodes(T.left) + countNodes(T.right)

return n

def Height(T):

#height of binary search tree

if T == None:

return 0

return 1 + max(Height(T.left), Height(T.right)) #max of either side is max of tree

def Embedding(T,k):

# Returns embedding of the word k is, or None if k is not in the tree

if T == None:

return None

if k == T.data.word:

return T.data.emb

elif T.data.word > k:

return Embedding(T.left,k)

return Embedding(T.right,k)

**------B-Tree------**

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

def FindChild(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 InsertInternal(T,i):

#Moves through tree to find respective leaf to append data to

if T.isLeaf:

InsertLeaf(T,i)

else:

k = FindChild(T,i.word)

if IsFull(T.child[k]):

m, l, r = Split(T.child[k]) #splits if found full

T.data.insert(k,m)

T.child[k] = l

T.child.insert(k+1,r)

k = FindChild(T,i.word)

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)

# Traverse through all array elements

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

# Last i elements are already in place

for j in range(len(T.data)-1-i):

# traverse the array from 0 to n-i-1

# Swap if the element found is greater

# than the next element

if T.data[j].word > T.data[j+1].word:

T.data[j], T.data[j+1] = T.data[j+1], T.data[j]

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 Insert(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.word)

InsertInternal(T.child[k],i)

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].word)

else:

PrintD(T.child[len(T.data)],space+' ')

for i in range(len(T.data)-1,-1,-1):

print(space,T.data[i].word)

PrintD(T.child[i],space+' ')

def countNodes(T):

#counts nodes in the tree

if T == None:

return 0

elif T.isLeaf:

return 1

else:

n = 1

for c in T.child:

n += countNodes(c)

return n

def Height(T):

#returns height of btree

if T == None:

return 0

elif T.isLeaf:

return 1

else:

return 1 + Height(T.child[0])

def Embedding(T,k):

# Returns embedding of the word k, or None if k is not in the tree

for i in T.data:

if k == i.word:

return i.emb

if T.isLeaf:

return None

return Embedding(T.child[FindChild(T,k)],k)

**------WordEmbedding------**

import numpy as np

class WordEmbedding(object):

#constructor

def \_\_init\_\_(self, word, embedding):

self.word = word #string for the word

self.emb = np.array(embedding, dtype=np.float32) #np array for embeddings

**VI – Academic Honesty Certification**

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.

x- Elisa Jimenez Todd