**CS 2302 Data Structures**

**Spring 2019**

**Lab Report #4**

Due: October 21, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were asked to take words from a text file as well as their embeddings and store them into a binary search tree or a B-Tree. Once these words and their embeddings were put into the data structure there was another function performed on them which measured the similarity between words. This was done using the embeddings as well. From this we are able to comprehend the efficiency and benefits associated with using either a binary search tree or a B-Tree.

**Proposed Solution Design and Implementation**

**Part #1**

For Part 1, I used a simple scanner which prompted the user to choose between option 1 or 2. These two options were for using a binary search tree or a B-Tree. The answer chosen was received as an integer and then placed in if-statements to determine what data structure would be used.

**Part #2**

To read the files I used two functions that I found while researching possible solutions to this lab. One function is called readData which reads the text file that has the word with its associated embedding. The other function used is ReadPairs which is only used on reading the text file that contains the words that will be compared to measure their similarity. The WordEmbedding object was difficult to make and I ended up implementing a very broad constructor that did not perform as required. Each problem with the WordEmbedding object was case specific to the data structure I was trying to implement.

**Part #3**

To compute the similarities between the words I used a function called CalSimialrity which used both numpy functions as described in the Lab Assignment. The issue I ran into was how to determine the number produced by these numpy functions. It would seem like these functions would produce float values, but my compiler kept saying that they were unsupported operand types. I could not solve this problem.

**Experimental Results**

**﻿Choose table implementation.**

**Type 1 for binary search tree or 2 for B-Tree**

**Choice: 2**

**Maximum number of nodes: 5**

**Building BTree:**

**stats:**

**Number of nodes: 70591**

**Height: 8**

**Running Time for construction: 2.2256410121917725**

**Word similarities found:**

Running Time

**Conclusion**

In conclusion this lab taught me of how simple modifications to code can change the time complexity from linear to constant time. This is clearly seen through the difference between the List class and the SortedList class. The main differences are seen when trying to access a certain element that has a specific quality. When trying to find the minimum element in a normal list you have to iterate and compare through each element. When the list is sorted the smallest element will always be at the head. This type of modification has shown me that minor changes to code can drastically make your program run more efficiently.

**Appendix**

**﻿**

**﻿ ﻿from timeit import time as time**

**import codecs**

**import numpy as np**

**from BTrees import BTree**

**from BST import BSTLab**

**from WordEmbedding import WordEmbedding**

**def readData(filename):**

**data = []**

**with codecs.open(filename,"r","utf\_8\_sig") as f:**

**for line in f:**

**chunks = line.split()**

**word, embedding = chunks[0], np.array([float(value) for value in chunks[1:]])**

**data.append(WordEmbedding(word, embedding))**

**return data**

**def ReadPairs(filename):**

**data = []**

**with codecs.open(filename,"r","utf\_8\_sig") as f:**

**for line in f:**

**data.append(tuple(map(lambda x: x.strip(), line.split(","))))**

**return data**

**def CalSimilarity(e1,e2):**

**return np.dot(e1,e2) / ((np.linalg.norm(e1) \* np.linalg.norm(e2)))**

**if \_\_name\_\_ == "\_\_main\_\_":**

**TxtFilename = "glove.6B.50d.txt"**

**pairFilename = "pair.txt"**

**print ('Choose table implementation.')**

**print ('Type 1 for binary search tree or 2 for B-Tree')**

**word = int(input('Choice: '))**

**if (word == 1):**

**T = BSTLab()**

**print ("Building BST: ")**

**tempFile = readData(TxtFilename)**

**start = time.time()**

**for x in tempFile: T.Insert(x)**

**end = time.time()**

**total = (end - start)**

**print("stats: ")**

**print("stats: ")**

**print("Number of nodes: ", T.NumItems())**

**print("Height: ", T.Height())**

**print("Running Time for construction: ", total)**

**elif (word == 2):**

**n = int(input("Maximum number of nodes: "))**

**T = BTree([],n)**

**print ("Building BTree: ")**

**tempFile = readData(TxtFilename)**

**start = time.time()**

**for x in tempFile: T.Insert(x)**

**end = time.time()**

**total = (end - start)**

**print("stats: ")**

**print("Number of nodes: ", T.NumItems())**

**print("Height: ", T.Height())**

**print("Running Time for construction: ", total)**

**else:**

**print ('Incorrect input')**

**pair = ReadPairs(pairFilename)**

**print ("Word similarities found: ")**

**total = 0**

**for (x0,x1) in pair:**

**start = time.time()**

**e1 = T.Search(x0)**

**e2 = T.Search(x1)**

**end = time.time()**

**total += (end - start)**

**if e1 == None or e2 == None: continue**

**print ("Similarity [{0},{1}] = {2:.4f}", CalSimilarity(e1,e2))**

**﻿import matplotlib.pyplot as plt**

**import numpy as np**

**class BSTLab(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)**

**elif T.data > newItem:**

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

**else:**

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

**return T**

**def search(node,k):**

**if node == None:**

**return**

**elif node.data == k:**

**return node**

**elif k > node.data:**

**return search(node.right,k)**

**elif k < node.data:**

**return search(node.left,k)**

**def Height(T):**

**if T == None:**

**return 0**

**lh = Height(T.left)**

**rh= Height(T.right)**

**return max([lh,rh])+1**

**def NumItems(T):**

**count = 1**

**if T.left != None:**

**count += NumItems(T.left)**

**if T.right != None:**

**count += NumItems(T.right)**

**return count**

**﻿class BTree:**

**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 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)**

**InsertInternal(T.child[k],i)**

**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 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]:**

**return i**

**return len(T.data)**

**def Split(T):**

**#print('Splitting')**

**#PrintNode(T)**

**mid = T.max\_data//2**

**if T.isLeaf:**

**leftChild = BTreeClass(T.data[:mid],max\_data=T.max\_data)**

**rightChild = BTreeClass(T.data[mid+1:],max\_data=T.max\_data)**

**else:**

**leftChild = BTreeClass(T.data[:mid],T.child[:mid+1],T.isLeaf,max\_data=T.max\_data)**

**rightChild = BTreeClass(T.data[mid+1:],T.child[mid+1:],T.isLeaf,max\_data=T.max\_data)**

**return T.data[mid], leftChild, rightChild**

**def IsFull(T):**

**return len(T.data) >= T.max\_data**

**def Height(T):**

**if T.isLeaf:**

**return 0**

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

**def Search(T,k):**

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

**if k in T.data:**

**return T**

**if T.isLeaf:**

**return None**

**return Search(T.child[FindChild(T,k)],k)**

**﻿import numpy as np**

**class WordEmbedding(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)**

**def \_\_gt\_\_(self,other):**

**if isinstance(other, WordEmbedding): return self.word > other.word**

**return self.word > other**

**def \_\_lt\_\_(self,other):**

**if isinstance(other, WordEmbedding): return self.word < other.word**

**return self.word < other**

**def \_\_eq\_\_(self,other):**

**if isinstance(other, WordEmbedding): return self.word == other.word**

**return self.word == other**

**def \_\_repr\_\_(self):**

**return self.word**

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