Understanding basic concept of Natural Language Processing using Hash Tables and BSTs

Laboratory #5

Ivan Gastelum

CS 2302 Data Structures

Dr. Olac Fuentes

TA: Nath Anindita

April 9th 2019

**INTRODUCTION:**

Natural Language Processing (NLP) is the sub-field of artificial intelligence that deals with designing algorithms, programs, and systems that can understand human languages in written and spoken forms. Word embeddings are a recent advance in NLP that consists of representing words by vectors (or arrays) of floating point numbers in such a way that if two words are similar, their embeddings are also similar. See https://nlp.stanford.edu/projects/glove/ for an overview of this interesting research.

In this laboratory project, we were assigned to download a txt file from the website mentioned above which contains 400,000 words, and each word with its embeddings in the form of an array of 50 float values. The objective is to read the big file and store each word along with its embeddings into two types of data structures: Binary Search Tree, and Hash Table that solves collisions through chaining. While performing the storing functions, we will have to record time and check which way of constructing the data structures results to be more efficient.

In addition to their efficiency, some other stats such as standard deviation, load factor, depth and other related to each data structure needed to be calculated. After calculating this, the data structure will be used to perform a similarities check with the help of the embeddings stored in the word object created for each actual word in the file. By doing this we were exposed to the beginning of understanding basic concepts and theory behind NLP and how a system like Siri or Alexa actually works.

The following instructions on how the code should work:

*1. Prompt the user to choose a table implementation (binary search tree or hash table with chaining).*

*2. Read the file ”glove.6B.50d.txt” and store each word and its embedding in a table with the chosen implementation. Each node in the BST or hash table must consist of a list of size two, containing the word (a string) and the embedding (a numpy array of length 50). For the hash table, choose a prime number for your initial table size and increase the size to twice the current size plus one every time the load factor reaches 1. Caution: do NOT recompute the load factor every time an item is entered to the table, instead, add a num items fields to your hash table class.*

*3. Compute and display statistics describing your hash table. See the appendix for examples for both implementations. Feel free to suggest others.*

*4. Read another file containing pairs of words (two words per line) and for every pair of words find and display the ”similarity” of the words. To compute the similarity between words w0 and w1, with embeddings e0 and e1, we use the cosine distance, which ranges from -1 (very different) to 1 (very similar), given by:*

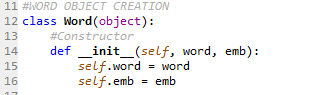
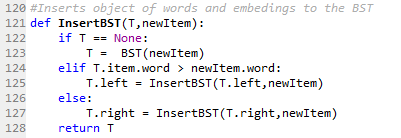
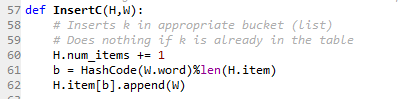
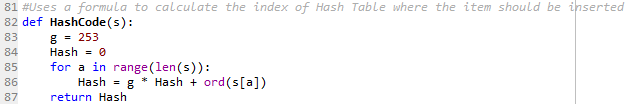
*where e0 · e1 is the dot product of e0 and e1 and |e0| and |e1| are the magnitudes of e0 and e1.*

*5. Display the running times required to build the table (item 2) and to compute the similarities (item 4).*

*Do not include the time required for displaying results. Use a large enough word file for item 4 in order to derive meaningful results.*

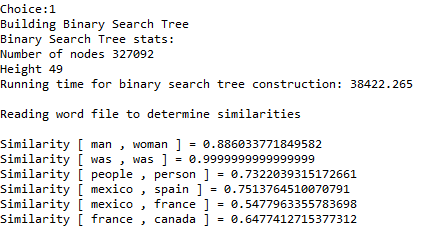
**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:**

Solutions to problems 1 – 5:

1. Prompting the user to choose either the creation of BST or Has Table can be designed with an input answer, or two conditionals where each different code will be run depending the user’s input. See code in appendix.
2. Inside these two conditionals, the work is divided into two different ways of inserting each word object into a BST or Hash Table. But both codes are similar in how they will read the txt file. Since we have two types of values in the file (the word, and array of floats) each line should be read and break down both of these types of data that would eventually become attributes of an object. See below the constructor of the Word Object:   
     
     
     
   So, each word along with its embeddings will be stored in an object. But the in order to do that, an array of objects need to be created in order to store each attribute as each line in the file is read.   
     
   Binary Search Tree solution:  
   The insertion of a BST is simple; if the BST is empty, the first node will be the root, then the next will be inserted depending on a comparison of each value to the next object, in this case we are comparing the words attained and converted into string in alphabetical order, doing so, the BST is generated, and by the time it ends, the BST will have the strings in alphabetical order. See below insertion function:  
     
     
   Hash Table solution:  
   The insertion method of Hash Table is different from a BST, but definitely more efficient in terms of time as it theoretically its running time is O(constant) as opposed of a BST which is O(log n). Since a Hash table with chaining is a list of many lists inside usually named ‘buckets’ there has to be a way of intelligently place the word in the right bucket so that the has table can be balanced in a way of its load factor and the percentage of empty buckets. As a result, a method was designed to attain a ‘Hash code’ which will represent the index of the bucket where that word should be positioned. See below insertion method with its Hash Code:  
     
   
3. Stats are used to check some properties of the data structure generated.   
   For BST:  
   Number of Nodes, Height, and Running Times  
   For Hash Table C:  
   Initial and final size, percentage of empty lists, standard deviation in lengths of buckets, and the load factor. See appendix for code of each function.
4. Right after the creation of the data structure, another file is created where two words are written at each line, and the program calculates the similarity from a range of -1 to 1. The approach was to follow the formula shown above, with its embeddings. The only limitation is that the words should be in the big file, so we can access to its embeddings and calculate the similarity. Otherwise the program won’t be able to perform the calculation. When one word is typed into the second file, the program traverses through the data structure and access to its embeddings.

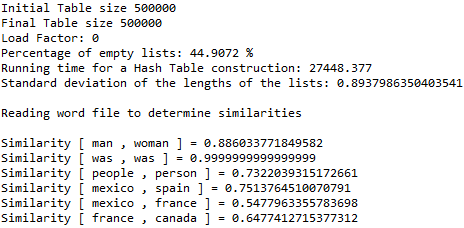
**EXPERIMENTAL RESULTS:**

**Choice 1: Binary Search Tree BST generation**



When we choose to generate the Binary Search Tree and store the data, we get something as shown above, it will depend on the words you type in the second file. In this case woman and man seem to have a high similarity, which makes sense since both are genres, I typed two exact same words in order to check the veracity of the program itself, it calculates to be 0.9999 in similarity, in theory it should give a whole number of 1.0, so we can say that it has some minimal error in the calculations. The BST running time was: 38,422 milliseconds, which is a good time, considering that it stored 400,000 words and ignore other words that did not start with a letter.

**Choice 2: Hash Table with Channing**



By looking at the Hash Table version of the similarities, the values are the same value, which makes sense that the word is found. In terms of running times, it looks like it follows the conclusion that Hash Table are in fact much more efficient than a BST, considering the fact that O(log n) is actually a very efficient time in data structure.

**CONCLUSIONS:**

Theoretically Hash table has a running time of constant, resizing method might take longer to rebuild a hash table and insert all numbers, but it can still be more efficient than a Binary Search Tree. Most problems of insertion can be solved using Hash Tables, and when it comes with big data, artificial intelligence, in order to reduce the time at its best and lowest, a Hash Table is the best and first choice to tackle a coding problem.

**APPENDIX:**

