CS2302 - Data Structures
Spring 2019
Lab # 5
Deadline: Monday, April 1, 11:59 p.m.

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 arryas) 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 order to work in real-time, NLP applications such as Siri and Alexa use hash tables to efficiently retrieve the embeddings given their corresponding words. In this lab you will implement a simple version of this. The web page mentioned above contains links to files that contain word embeddings of various lengths for various vocabulary sizes. Use the file *glove.6B.50d.txt* (included in the file *glove.6B.zip*), which contains word embeddings of length 50 for a very large number of words. Each line in the file starts with the word being described, followed by 50 floating point numbers that represent the word's vector description (the embedding). The words are ordered by frequency of usage, so "the" is the first word. Some "words" do not start with an alphabetic character (for example "," and "."); feel free to ignore them.

Your task for this lab is to compare the running times of two implementations of tables to retrieve word embeddings to enable the (hopefully fast) comparison of two given words. One of your table implementations should use a binary search tree; the other should use a hash table with chaining. See the appendix for sample runs of your program.

Your program should do the following:

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 w_0 and w_1, with embeddings e_0 and e_1, we use the cosine distance, which ranges from -1 (very different) to 1 (very similar), given by:
$$sim(w_0, w_1) = \frac{e_0 \cdot e_1}{|e_0||e_1|}$$
where $e_0 \cdot e_1$ is the *dot product* of e_0 and e_1 and $|e_0|$ and $|e_1|$ are the magnitudes of e_0 and e_1.
Recall that the dot product of vectors u and v of length n is given by $u \cdot v = u_0 * v_0 + u_1 * v_1 + \ldots + u_{n-1} * v_{n-1}$ and the magnitude of a vector u of length n is given by $|u| = \sqrt{u \cdot u} = \sqrt{u_0^2 + u_1^2 + \ldots u_{n-1}^2}$.
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.

Use the program *hash_table_chain_strings.py* provided in the class webpage as a starting point for your hash table implementation. It shows how to compute hash values for strings and how to include multiple fields in a hast table node.

As usual, write a report describing your work. Discuss the efficiency of the two methods being compared and also comment on word embeddings as a way or representing words.

Appendix

Sample run 1:

```
Choose table implementation
Type 1 for binary search tree or 2 for hash table with chaining
Choice: 1

Building binary search tree

Binary Search Tree stats:
Number of nodes: XXX
Height: XXX
Running time for binary search tree construction: XXX

Reading word file to determine similarities

Word similarities found:
Similarity [bear,bear] = 1.0000
Similarity [barley,shrimp] = 0.5353
Similarity [barley,oat] = 0.6696
Similarity [federer,baseball] = 0.2870
Similarity [federer,tennis] = 0.7168
Similarity [harvard,stanford] = 0.8466
Similarity [harvard,utep] = 0.0684
Similarity [harvard,ant] = -0.0267
Similarity [raven,crow] = 0.6150
Similarity [raven,whale] = 0.3291
Similarity [spain,france] = 0.7909
Similarity [spain,mexico] = 0.7514
Similarity [mexico,france] = 0.5478
Similarity [mexico,guatemala] = 0.8114
Similarity [computer,platypus] = -0.1277

Running time for binary search tree query processing: XXX
```

Sample run 2:

```
Choose table implementation
Type 1 for binary search tree or 2 for hash table with chaining
Choice: 2

Building hash table with chaining

Hash table stats:
Initial table size: XXX
Final table size: XXX
```

```
Load factor: XXX
Percentage of empty lists: XXX
Standard deviation of the lengths of the lists: XXX

Reading word file to determine similarities

Word similarities found:
Similarity [bear,bear] = 1.0000
Similarity [barley,shrimp] = 0.5353
Similarity [barley,oat] = 0.6696
Similarity [federer,baseball] = 0.2870
Similarity [federer,tennis] = 0.7168
Similarity [harvard,stanford] = 0.8466
Similarity [harvard,utep] = 0.0684
Similarity [harvard,ant] = -0.0267
Similarity [raven,crow] = 0.6150
Similarity [raven,whale] = 0.3291
Similarity [spain,france] = 0.7909
Similarity [spain,mexico] = 0.7514
Similarity [mexico,france] = 0.5478
Similarity [mexico,guatemala] = 0.8114
Similarity [computer,platypus] = -0.1277

Running time for hash table query processing: XXX
```