Siri Team Assignment
RhymeEngine
Prof. Young
Apple

Pawan Rajpoot
Teammates: Individual
Assignment Date: 04/07/18
Submission Date: 13/07/18

Problem Statement

Design and Code a rhymeEngine which outputs closest rhyming word for the input word with reference to a given text vocabulary.

Initial Solution

Initially I came up with a pretty straight forward solution.

- reverse all words in vocab.
- pad them so as to make every string of 20 letters(max length).
- sort the above list.
- do a binary search for the input word in sorted list.
- return +3 and -3 words from list.
- return closest word from list above.

Problem in above solution is the assumption that the word is present in the vocabulary. Also for abrupt neighbors suffix need not to match(e.g. for original words **form** and **ten** we get sorted vocab as **mrof** – **net#**)

Investigation/Research

Main objective was to somehow represent the words in a N dimension vector (N is the max letter word) so that nearest neighbors can then be calculated for the given input. Nearest neighbor Search is itself a complex optimization problem.

Quoted from wiki – "as the curse of dimensionality states that there is no general-purpose exact solution for NNS in high-dimensional Euclidean space using polynomial preprocessing and polylogarithmic search time."

So I devised a simpler straight forward solution where in I index the words using tries and then do a search on them accordingly.

Indexing words on letters, so that search with respect to suffix can be easier was an obvious next step. Also, read about Soundex and Metaphone phonetic algorithms which are used for indexing words according to English pronunciation.

Why Trie

Why Trie? :- With Trie, Insertion and finding strings is done in O(L) time where L represent the length of a single word. No collision handling is required (like we do in open addressing and separate chaining) Words can be printed in alphabetical order which is not easily possible with hashing. We can efficiently do prefix search (suffix in this case) with Trie.
Issues :- Tries needs lot of overhead memory.

Solution 2

Taking inspiration from the above first immediate solution, and trie based indexing.

- reverse all words in vocab.
- create a trie.
- for a input word reverse the word, search for prefix of reversed word in incremental order.
- for the longest positive prefix return all the words from trie.
- return any one random from above list.

Problem in this solution is inability to deal with words where suffices have similar pronunciation but they spell differently like **Smith** and **myth**.

Solution 3

To deal with the problem of solution 2 I incorporated Soundex in it. Followed following steps:

- get soundex of each word.
- create a map of key soundex and value all words.
- reverse all words in soundex vocab.
- create a trie for soundex reversed vocab.
- for a input word, get soundex, reverse the soundex, search for prefix of reversed word in incremental order.
- for the longest positive prefix return all the words from trie.
- return any one random from above list, get value list for that key in soundex map, return any one.

Problem in this solution is Soundex is very weak to capture the similar pronunciation. Like for words "lil" and "lol" both have same Soundex L400, this created problem in the outcomes. I also tried with "Metaphone" algorithm instead of "Soundex", which is supposedly a improved version of it, still it had similar kind of problems in capturing correct phonetic suffices. **Although** if we get a better phonetics algorithm this solution will create optimum results.

Analysis/Complexity

Solution 1
Sorting takes O(nlogn)
Binary search takes O(logn)
Solution 2
Creating trie If you store all possible pointers to other nodes (all C of them, where C is your constant, which is the number of distinct letters), then yes, the space complexity is O(number of nodes*C). However, if we store a re-sizable array of pointers to other nodes, then the number of pointers in total would be O(number of nodes), and the amount of space you give them is also O(number of nodes).)

Search on trie: Same goes for looking up words later: you perform C steps for each of the W words.

Solution 3

Indexing and searching similar to solution 2, Soundex hashmap will take N extra memory in worst case. Soundex takes O(N*M) time.

Time spent

Solution 1: Completed in half hour.
Solution 2: Completed in about one and a half an hour.
Solution 3: Added code to solution #2 in half an hour.
Research: Spend around 2 hours on reading about Nearest Neighbor Search, Soundex, Metaphone and Curse of Dimensionality.

Future Work

Solving the problem using the initial frame of mind, that is solving it as a Nearest Neighbor Search.

Java Imp

Soundex.java: class with method "getShortcut" which returns soundex code for a string.
TrieNode.java: Basic Data Structure for a trie Node. Contains an array of 26 length for storing subsequent letters in a word.
TrieUtils.java: Functions to create, search, return node and return List of possible words for a prefix.
TestTrie.java: Entry point, main function to start the project.
Note: Libraries Used: Only native Java, nothing extra.

Attachments

Zip file with Java implementation of Solution 2.
Files for Solution 3 are included.
How to run code pdf.

References

[1] vocab used:https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-no-swears.txt

[2] https://www.genealogieonline.nl/en/naslag/phonetic/.

[3] https://en.wikipedia.org/wiki/Metaphone

[4] High-dimensional approximate nearest neighbor: k-d Generalized Randomized Forests, Yannis Avrithis, Ioannis Z. Emiris, Georgios Samaras