Mobile Device Keyboard

The purpose of this application is to learn the words typed by the user over time, and then when prompted will output a ranked list of auto-complete candidates. Furthermore, there are two different modes for this application. First being training and this is where the user enters words and phrases, so the algorithm can learn word preferences. The next mode is where the use enters a word fragment and in return receives a ranked list of possible word choices based on most frequently used words.

Ex:
Training data: Johny picked a pipe of peppers, plus a pack of pickles.
Input: “p” -> [“picked” (1), “pipe” (1), “peppers” (1), “plus” (1), “pack” (1), “pickles” (1)]

How To Run

1. First, we need to make sure we have Java downloaded.
2. Next you need to download the files locally from the github source. Once you have installed the files on your local machine open your terminal if you have not already, and change directories into the directory you saved the project files.
3. Then you can compile the using the command javac Runner.java.
4. Now we should be able to run the program using the command java Runner.
5. Now that the program is running you can enter train to enter training data into the algorithm, type to enter a word fragment and receive suggestions or enter exit to quit.

Implementation Details

For this project I chose to use a hash map for the single reason that they have extremely efficient lookup time as long as there are not too many collisions in the table. This is beneficial because once we have trained the data, and the user enters a word fragment we can retrieve that list in O(1) time. The implementation with the hashmap does have relatively good runtime compared to other data structures, however, it does require a little more space. An alternative approach would have been to use a trie to store the suggested words, but at the end of the day the hashmap seemed to be a good choice for its fast lookup speed. Another implementation detail is that after we update the confidence the candidate is swapped until it’s in its correct position. A different method would be to use a binary search to find the correct position for the candidate. This seems to only make a significant difference if the lists grow very large because when we update the confidence we only increment by one so on average we only need to swap the candidate up one rank at a time.

Testing

For testing purposes there is a file called TestFile.java. This file contains multiple test cases that test the basic functionality that Asymmetrik asked for. Other test include testing strings with punctuation, testing very long strings with multiple common words and testing the empty string and numbers. Lastly, if the user would like to add their own test cases all they have to do is add them to TestFile.java, and then call the function in the constructor.

Final Remarks

If anyone using this application has any questions feel free to email me at paulhendriksen@yahoo.com (mailto:paulhendriksen@yahoo.com). Thanks!