Programming Assignment for Module 3, Part 2: Markov Text Generation

Let's have some fun. In this assignment, you'll implement Markov Text Generation using a List of Lists.

Before you begin this assignment, make sure you check Part 2 in the setup guide to make sure the starter code has not changed since you downloaded it. If you are an active learner, you will have also received an email about any starter code changes. If there have been any changes, follow the instructions in the setup guide for updating your code base before you begin.

1. Find the starter code

You should see a package called textgen in that starter code.

To verify everything is setup okay, you can run the "MarkovTextGeneratorLoL.java" file and you will see output (including two "null" outputs which we'll be fixing soon!).

2. Open and examine the starter code

Examine the **MarkovTextGenerator** interface for the key methods:

public void train(String sourceText);

Markov Text Generation depends on an initial source text to mimic. This method takes in String to train the Markov Text Generator.

public String generateText(int numWords);

The goal of Markov Text Generation is to be able to generate text which resembles the source in a reasonable way. The method generateText returns such text as a String of words, the length of which is determined by numWords.

public void retrain(String sourceText);

You may wish to use a Markov Text Generator multiple times with different source text. The method retrain acts just like train, except it removes the existing training before

Assignment and Submission Details

Your submission of this assignment will be the code which implements Markov Text Generation as a List of Lists. **As always, we recommend testing as you write your implementation.**

Step 1: Implement the train method

You'll notice our MarkovTextGeneratorLoL constructor creates a List of ListNode objects. The ListNode class is authored at the bottom of the MarkovTextGenerationLoL.java file. Each ListNode contains a word and a list of words which follow that word in the source text. We'll be using ListNodes to help us generate text (in the next step).

1. A. The algorithm you'll be implementing

The idea for the train method is to build your list of lists:

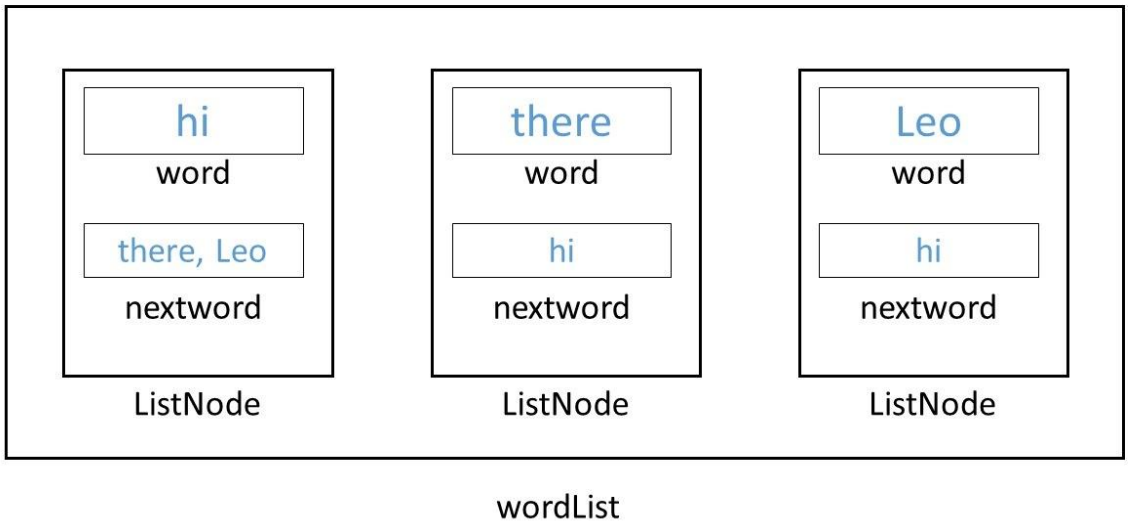
```
set "starter" to be the first word in the text
set "prevWord" to be starter
for each word "w" in the source text starting at the second word
   check to see if "prevWord" is already a node in the list
      if "prevWord" is a node in the list
         add "w" as a nextWord to the "prevWord" node
      else
         add a node to the list with "prevWord" as the node's word
         add "w" as a nextWord to the "prevWord" node
    set "prevWord" = "w"

add starter to be a next word for the last word in the source text.
```

First, check to see if you understand how this algorithm works. You'll be going through the text keeping track of the previous word you saw ("prevWord") and the current word "w". You'll then want to add this current word to the list of words which follow the previous word. It's okay to add a word multiple times to the list of words which follow - we'll use this to help with our text generation (words appearing more than once should be more likely to occur). Notice that there is extra care taken to setup the "starter" word and to make sure the last word also points to the start word.

An example of how the train function should work can be found below. Notice that starter will point to "hi".

The wordlist contents (below) if trained on the string:
hi there hi Leo



1. B. Use the algorithm above to author your train method

You'll find that the list itself has already been made for you and the methods you need in a ListNode also already exist. Your code then will be searching the list of ListNodes, calling ListNode constructors to add previous words to the list, and adding following words using the addNextWord method.

After completing your train method, we recommend you use the toString methods in both MarkovTextGeneratorLoL and ListNode to verify you are producing a reasonable list. For example, if you train your generator on the string above ("hi there hi Leo"), calling toString on the MarkovTextGenerator should produce:

hi: there->Leo->

there: hi->

Leo: hi->

Step 2: Implement the generateText method

Now that you've trained on text, your next step will be producing text based on the input set. To do this, you'll be implementing the algorithm below.

```
set "currWord" to be the starter word
set "output" to be ""
add "currWord" to output
while you need more words
   find the "node" corresponding to "currWord" in the list
   select a random word "w" from the "wordList" for "node"
   add "w" to the "output"
   set "currWord" to be "w"
   increment number of words added to the list
```

To help with implementing the method, you should author the ListNode getRandomNextWord method to help. It will take care of the step above:

```
select a random word from the "wordList" for "node"
```

To clarify how this algorithm behaves, let's continue with the example above (trained on "hi there hi Leo") and generate 4 words.

We add "hi" to the output as the starter word.

We then find the node corresponding to the word "hi". "hi" could generate either "there" or "Leo". Let's suppose the random value selects "Leo" and we add "Leo" to our output.

"Leo" can only generate "hi" so we add "hi" to our output.

Again, "hi" could generate either "there" or "Leo". Let's suppose the random value selects "there". So we add "there" to our output.

We've added 4 words to the output ("hi", "Leo", "hi", "there") so we're done (our while loop would terminate). Our final output would be:

"hi Leo hi there"

NOTE: If the generator has not yet been trained, the generateText method should simply return an empty list. Normally you would also print a warning message, but our auto grader might get confused so just return the empty list without printing anything.

Step 3: Implement the retrain method

Retrain will behave just like train, only you'll need to re-initialize the instance variables, effectively discarding the prior training.

Hints:

- You will likely find the "toString" method for the list and/or a "toString" method for each node helpful when testing and debugging.
- Train on small inputs and draw the expected list of lists by hand. Then check to see if your code is producing what you'd expect.
- When picking a random next word, be sure you are able to produce all of the possible words. For example, if you could produce "hi", "there", or "this", be sure your getRandomNextWord method can produce all possible words. A common mistake is to not bound your random number properly and go off the end of the list or omit the last word in the list.
- Also, when picking a random next word, be sure to have a test case where a word is repeated. For example, the nextWord list could contain "hi", "hi", "hi", "hello". Your generate next word method should produce "hi" far more often than it produces "hello" if you have it produce a reasonable (10+) number of words.
- Punctuation counts, so it's okay if you end up with the same word in your data structure punctuated and not punctuated. For example, if you trained on the following: "Hi there. Up there is the sky." You would have a node for "there." and a node for "there" in the resultant List of Lists.

What and how to submit

Upload the MarkovTextGeneratorLoL.java file for automated testing. We'll be testing to ensure your output is reasonable: words only follow words as in the training text and all possible subsequent words get generated at a reasonable frequency (i.e., the random logic works correctly).