Homework 10: Text Generation / Stage III

In Stage I of this project we built the basic infrastructure for a machine learning program designed to generate random texts that look like they were written by human authors. In Stage II we implemented a training algorithm that "learns" the style of a given corpus. In Stage III we'll implement a text generation algorithm that can create endless random texts written in the learned style.

Before we go on, a word of caution about the so-called "learning" and "generation" processes that we are implementing in this project. In Stage II, the curious student may have thought that the learning algorithm will learn sophisticated linguistic aspects like sentence composition etc. In fact, we learned nothing more than the sequential patterns in which *characters* are arranged in a given corpus. It is quite shocking and somewhat embarrassing that that's all that is needed for randomly generating reasonably good looking texts, as we now turn to describe. We'll start with the basic task of generating random characters.

III.1 Generating random characters from a given text

Close your eyes, point at the screen, then open your eyes and look at the screen area around your finger. Suppose that you run this experiment many times. Quite likely, most of the time your finger will be closer to an '`e`' than to a '`q`'. That's because in English texts '`e`' occurs more frequently than '`q`'. This is an example of a property that our random text generation approach must preserve: in order for the generated text to "look good", the character frequencies in it must be similar to the character frequencies in the training text. Of course, this is only a necessary condition, and not a sufficient condition, for generating good looking texts.

We already know how to analyze, and record, the character frequencies in a given text. We did it in stages I and II. For example, here is the list of `CharData` objects associated with the input `"committee "` (the last character in this input is space):

```
((_ 1 0.1 0.1) (e 2 0.2 0.3) (t 2 0.2 0.5) (i 1 0.1 0.6) (m 2 0.2 0.8) (o 1 0.1 0.9) (c 1 0.1 1.0))
```

The list contains all the unique characters in the input, in no particular order. Each list element records a character (`chr`), how many times the character appears in the input (`count`), the character's frequency, or probability (`p`), and cumulative probability (`pp`). For ease of reference in what follows, the `chr` and `pp` values are underlined.

How can we generate random characters from the text `"committee "`? We can do it using the above list, and the *Monte Carlo* technique described in lecture 5-1, slides 19-21. We start by drawing a random number from the range [0,1). Let's call the resulting value *r*. We then iterate through the list, reading the cumulative probabilities (`pp` values) as we go along. We stop at the `CharData` element whose `pp` value is greater than *r*, and return the character of this element. For example, if *r* = 0.38 we return '`t`'; if *r* = 0.72 we return '`m`', and so on. Note that the maximal *r* that we can possibly get is 0.99999…, in which case we return (in this particular case) the character '`c`'.

The Test2 class (revisited)

In Stage I of this homework you completed the code of several functions (static methods) whose signatures were given in a class named `Test2.java`. At this stage you are required to add two more functions to this class. Start by creating a backup copy of the `Test2.java` file that you've already completed. Next, complete the code in the `Test2.java` supplied in stage III.

Implementation notes

Implement the `Test2` functions in the order given below.

`buildList:` You've implemented this function in Stage I. Copy-paste your code here.

`calculateProbabilities:` You've implemented this method in Stage I. Copy-paste your code here.

`getRandomChar(list):` This function starts by generating a random number in the range `[0,1)`. Let's call this value `r`. The function then uses an iterator to go through all the `CharData` elements of the given `list`. The function stops the iteration when the `pp` value of the current element is greater than r, and returns the `chr` value of that element.

`testRandomCharGeneration(input, T):` This function is designed to stress-test the `getRandomChar` function. Read the given code and comments, and complete the function's implementation.

Interesting observation: The relative frequencies of characters in a string are independent of the order of characters in the string. For example, the two strings `"william shake speare"` and "`iam aweakish speller`" are permutations of each other, and thus must have the same character frequencies.

This begs the following question: If we apply the sequential Monte Carlo technique described above to generate random characters from two permutations of the same string, are we guaranteed to get *the same statistical results*? Remarkably, the answer is yes. In order to test this hypothesis, the `main` function of `Test2` includes the following two tests:

```
testRandomCharGeneration("desserts", 100000);
testRandomCharGeneration("stressed", 100000);  // a permutation of "desserts"
```

The two calls generate the same statistical results, as you can see below.

Output

Here is an example of running the new version of `Test2.java`:

```
% java Test2

Testing the construction of a list of CharData objects from a given string input.
The probability fields of the CharData objects will be initialized to 0.
Input = "committee "
List = ((  1 0.0 0.0) (e 2 0.0 0.0) (t 2 0.0 0.0) (i 1 0.0 0.0) (m 2 0.0 0.0) (o 1 0.0 0.0) (c 1 0.0
0.0))  (same output as in Stage I)
```

```
Testing the construction of a list of CharData objects from a given string input.
This time, the probability fields will be computed and set correctly.
Input = "committee "
List =  ((  1 0.1 0.1) (e 2 0.2 0.30000000000000004) (t 2 0.2 0.5) (i 1 0.1 0.6) (m 2 0.2 0.8) (o 1 0.1
0.9) (c 1 0.1 1.0))
```
(same output as in Stage I)

```
Testing the generation of random characters from the input "desserts":
Total number of trials: 100000
Number of trials that generated t: 12517
Number of trials that generated r: 12600
Number of trials that generated s: 37409
Number of trials that generated e: 24880
Number of trials that generated d: 12594
```
(t, r, and d have the same frequency X, e has frequency 2X, s has frequency 3X)

```
Testing the generation of random characters from the input "stressed":
Total number of trials: 100000
Number of trials that generated d: 12670
Number of trials that generated e: 25093
Number of trials that generated r: 12461
Number of trials that generated t: 12437
Number of trials that generated s: 37339
```
(likewise)

Task III

(Tasks I and II were specified in Stage I and Stage II, respectively). Complete the missing code in the given `Text2.java` file. The program outputs should be formatted like the example shown above.

III.2 Text Generation

Suppose we trained the `TextTrain` program from Stage II on the test file `sentence.txt`, using a window length of 2. This training will produce the map listed in Appendix II.6. Suppose we now want to use this map for generating text randomly. How can we go about it? An example is worth a thousand words, so here is one. The table below shows how the generated text could grow during the first few iterations of one particular text generation session:

> Generating random text that should be similar to the text in the training file `sentence.txt`. The generated text length will be 7 characters. The process starts with the given initial text "`hi`", and uses a window length of 2:

iteration	window	relevant list (taken from the map in II.6)	random number	selected letter	generated text (so far)
0	hi	((m 2 0.5 0.5) (n 2 0.5 1.0))	0.35	m	him
1	im	((s 1 0.5 0.5), (1 0.5 1.0))	0.42	s	hims
2	ms	((e 1 1.0 1.0))	0.98	e	himse

3	se	((l 1 1.0 1.0))	0.19	l	himsel
4	el	((p 1 0.5 0.5) (f 1 0.5 1.0))	0.86	f	himself

(The random numbers listed in the fourth column will vary from one text generation session to another, resulting with different generated texts)

Explanation: Starting with the initial window "`hi`", which is specified by the user, the text generator tries to get the value of the key "`hi`" from the map listed in Appendix II.6. The result is the linked list `((m 2 0.5 0.5)(n 2 0.5 1.0))`, from which the generator will randomly select one character. To do so, the generator draws a random number in the range `[0,1)`, getting, say, `0.35`. Following the Monte Carlo algorithm, the program selects the character '`m`', appends it to the end of the generated text, resulting with the string "`him`". The generator then moves the window to the last `windowLength` characters of the generated text, resulting with the window "`im`". The next iteration follows exactly the same logic: after getting from the map the list associated with "`im`", the program selects the character '`s`', and the generated text grows to become "`hims`". In the next iteration the window becomes "`ms`", and so on and so forth. We now turn to describe some operational details.

Initialization:

- If the length of the given initial text is less than the given window length, the generator prints the given initial text and terminates.
- The generator starts by setting the first window to the last `windowLength` characters of the given initial text.

Termination: The text generation process stops when one of two things happen:

- The length of the generated text so far equals the given text length, or
- The current window is not found in the map.

After stopping, the program prints the generated text.

III.3 TextGen

The text generation algorithm described above should be implemented in a program named `TextGen.java`. The `TextGen` program includes both the `train()` method developed in Stage II, and a method named `generate()` which is developed in this stage.

Handling Randomness

The testing of programs that use random number generation presents an operational challenge: *the program's behavior can't be replicated*. Each time you execute the program you get a different set of random numbers, and, as a result, a different program behavior. This is often a bonanza for the program's users, but a headache for the program's developer. For example, each time you run `TextGen.java`, the program generates a different random text. This is a lot of fun to watch. However, you cannot tell, with confidence, that your program is working properly. And, neither you nor us can run your code in a way that produces predictable results.

What is needed is an optional ability to create exactly the same sequence of random numbers, each time you execute your program. This can be done by using a random number generator that has a *seed* parameter (that's exactly what `Math.random` does, behind the scenes). The seed is an integer value which is used to initialize the function that generates random numbers. If you use the same seed value in different program executions, you will get exactly the same sequence of random values each time. Further, if you and I will use the same seed value on two different computers, both of us will get exactly the same sequence of random numbers. Mazel tov! We have a way to test your program systematically, and predictably, on any computer.

Java implements this capability using an object named `Random` (the `Math.random()` function constructs such an object, behind the scenes, and sets its seed to a random value, like some function of the current clock's value). Here is an example of using `Random` to generate random numbers with or without specifying a seed:

```
import java.util.Random;
...
// Initializing a random number generator that uses a random seed (which is the default):
Random randomGenerator = new Random();
// Each time we'll execute the program, the following statement
// will produce an unpredictable number between 0 and 1:
double random = randomGenerator.nextDouble();

...
// Initializing a random generator that uses a fixed seed, say 20:
Random randomGenerator = new Random(20);
// Each time we'll execute the program, the following statement
// will produce the same number between 0 and 1:
double random = randomGenerator.nextDouble();
```

The final version of `TextGen` should be able to operate in both of the above modes. As you will see in the next section and in the `main()` method code, we now allow the user to specify if he or she wants to execute the program using a random seed, or a fixed seed.

The program handles this requirement by creating, initializing, and using, a `Random` object which it calls `randomGenerator` (like the example above). Since the code that handles this headache is given, the only place in the program where *you* have to make a change is in your implementation of the `getRandomChar` method. In particular, your `getRandomChar` function probably makes a call to `Math.random()`. All you have to do is replace this method call with the method call `randomGenerator.nextDouble()`.

Usage

The text generation program expects to get five command-line arguments, as follows:

`% java TextGen` *windowLength initialText textLength* `fixed/random` *fileName*

For example, the following run generates a random Shakespearean text of length 1000 characters, styled after the play *Romeo and Juliet*, which is stored in the file `shake.txt`. The random text will start with the initial text `SAMPSON`, which is the name of one of the characters in the play:

```
% java TextGen 7 SAMPSON 1000 random shake.txt
```

```
SAMPSON.
Yes, madam!
Ay, let the potion’s force should disturb devotion!
Juliet, I will answer, I’ll no love thy conjure thee by Rosaline,
And your tears,
Which too untimely frost
Upon the thought long to woe,
Which she had laid it, and
put up my iron dagger.

FRIAR LAWRENCE.
Bliss be upon you for a holy marriage now: younger than my love? The all-seeing sun
Ne’er saw her mind early bid me
enquire you up, i’faith. Will it now.
```

… and so on and so forth, up to 1000 characters. I am not sure what William would have said about this text, but his publisher would probably publish it. Here is another run, using exactly the same command-line arguments:

```
% java TextGen 7 SAMPSON 1000 random shake.txt
```

```
SAMPSON.
Yes, madam; we have a soul of lead
So stakes me mad!
Day, night,
That I yet know it begins the world begun.

BENVOLIO.
Groan! Why, bride!
What, not a Montague.
What’s the furious eye doth lie a-bleeding,
My blood of our order,
I thought with hunt’s-up to the sunset of my joys are best.
```

… and so on and so forth. Maybe the publisher will take this one too. Another great writer was Ernest Hemmingway (Nobel prize, 1954). The following run generates a 45-character long text, starting with “`mustache`”, styled after Hemmingway’s short stories, which are stored in the file `hemm.txt`.

```
% java TextGen 8 mustache 45 fixed hemm.txt
```

```
mustaches
and big hands. Liz
was terribly frightened
```

Not sure what Ernest would have said about this text. Maybe kill himself (as he actually did).

If we’ll re-execute the program with exactly the same command-line arguments, we should get exactly the same output:

```
% java TextGen 8 mustache 45 fixed hemm.txt
mustaches
and big hands. Liz
was terribly frightened
```

So, that's one way to test that your program consistently: Execute it with a fixed seed. That's what we'll do when we test your submitted program.

Implementation notes

We now describe how to complete the code of `TextGen.java`.

Read the static variables declarations, the `main()` function, and the `init()` function, and make sure that you understand what these functions do.

`train:` You've implemented this function in Stage II. Copy-paste your code here.

`calculateProbabilities:` You've implemented this function in Stage I. Copy-paste your code here.

`mapString:` You've implemented this function in Stage II. Copy-paste your code here.

`generate:` Follow the example and explanation from section III.2, and write the code that implements them.

`getRandomChar(list):` You've implemented this function in Task III. Copy-paste your code here, and replace the call to `Math.random()` with a call to `randomGenerator.nextDouble()`.

Program structure

In order for `TextGen` and its tests to run, the following classes must be present in your working folder:

- `Node.java` (from Stage I)
- `CharData.java` (from Stage I)
- `List.java` (from Stage I)
- `ListIterator` (from Stage I)
- `StdIn.java` (given)
- `TextGen.java`
- `Shake.txt` (Romeo and Juliet corpus)
- `Hemm.txt` (Hemmingway's short stories corpus)

Take Home Lessons

This is the last homework in IDC's Introduction to Computer Science course. You've developed the ability to write complex multi-class object-oriented programs, using sophisticated data structures like linked lists and hash maps. We hope that you enjoyed the journey, and that you are eager and excited to learn more computer science and software engineering.

Submission

Zip the following files into `hw10.zip`:

- `List.java`
- `Test1.java`
- `Test2.java`
- `TextGen.java`

GETFEED will be available soon.

Deadline: Submit `hw10.zip` no later than January 22, 2021, 23:55.