Homework 9: Machine Learning, Stage III

You will be happy to learn that all the hard work was already done. If you've completed all the development and testing steps described in Stage I and Stage II of this project, then all you have to do now is implement the `generate` method, which is much simpler than the `train` method.

As we said previously, the text generation algorithm that we will use is shockingly effective and embarrassingly simple. The algorithm is described in the next section.

III.1 The text generation process

Suppose that during the training stage the program has learned the language model listed in Appendix II-A, with window length = 2. Suppose that we now tell the program to use this model to generate text randomly, starting with the initial text "hi".

The program will generate the text randomly, and gradually, using a loop. This generative strategy is sometimes referred to a *Markovian process*. An example is worth a thousand words, so here is one. The table below shows how the generated text evolves during the first 5 iterations of one particular program execution. (The random numbers will vary from one program execution to another, resulting with different generated texts.)

Iteration	Window	List	Random number	Generated letter	Generated text (so far)
0	hi	((n 2 0.5 0.5) (m 2 0.5 1.0))	0.71	m	him
1	im	((1 0.5 0.5) (s 1 0.5 1.0))	0.66	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

Explanation: Starting with the initial window "hi", which is specified by the user, the program will try to get the value of the key "hi" from the map (which is listed in Appendix II-a). The result will be the linked list `((n 2 0.5 0.5)(m 2 0.5 1.0))`. The program will then draw a random number, which happens to be 0.71. Since the relevant cumulative probability distribution is `((n 0.5)(m 1.0))`, the program will select the character 'm', and append it to the end of the generated text string. The result will be the string "him". The system will then move the window to the last `windowLength` characters of the generated text, resulting with the window "im". The next iteration will follow exactly the same logic: after getting from the map the list associated with "im", the program will randomly select the character 's', and the generated text will grow to become "hims". In the next

iteration the window will become "ms", and so on and so forth. We now turn to describe some operational details.

Note that in the text generation stage, as in the training stage, we use a fixed window length, as specified by the user. Remember that we keep this value in a property (field) of the `LanguageModel`, named `windowLength`.

If the length of the initial text (as specified by the user) is less than the `windowLength`, we cannot generate any text. In this case we return the initial text, and terminate.

The text generation process starts by setting the initial window to the last `windowLength` characters of the initial text supplied by the user. In each iteration, the window is set to the last `windowLength` characters of the text that was generated so far.

The text generation process stops when the length of the generated text equals the desired text length, as specified by the user.

In any iteration, if the current window is not found in the map, we stop the process and return the text that was generated so far.

III.2 Implementation

The process described above should be implemented in the `generate` method. This method takes two parameters: `initialText` (a string), and the desired `textLength` (an integer) of the generated text.

Write the code of the `generate` method, following the logic described in the previous section.

III.3 Handling randomness

There is an inherent problem in developing and testing programs that use Java's `Math.random` function: *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.

In the context of this particular program, the implication is that each time you'll run the program, the program will generate a different random text. This will be a lot of fun to watch. However, you will not be able to tell, with complete confidence, that your program is working properly. And, neither you nor us will be able to run your code in a way that produces replicable results. In theory as well as in practice, experiments that cannot be replicated are not worth much.

What is needed is an option to create exactly the same sequence of random numbers, each time you execute your program.

This can be done by specifying a *seed*. The seed is a 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 controlled random numbers generation capability using a class called `Random`. Here is an example of using `Random` to generate random numbers with or without specifying a seed:

```
// Initializing a Random object (from Java's Random class)
// without specifying a seed:
Random randomGenerator = new Random();
// Each time we'll execute the program, the following statement
// will produce some unpredictable random number between 0 and 1:
double random = randomGenerator.nextDouble();


// Initializing a Random object (from Java's Random class), using a seed:
Random randomGenerator = new Random(20);  // seed = 20
// Each time we'll execute the program, the following statement
// will produce the same random number between 0 and 1
double random = randomGenerator.nextDouble();
```

We wish our final version of the `LanguageModel` class to support both modes of operation. In order to do so, we add to the `LanguaheModel` class a field that refers to a `randomGenerator` object, and two class constructors. One constructor is designed to create a `LanguageModel` object with a random seed, and the other constructor is designed to create a `LanguageModel` object with a fixed seed = 20 (the specific number is not important, as long as we all agree to use the same number).

As you will see in the usage section (below) and in the supplied `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.

Since the code that handles all of the above headache is given to you, the only place in the program where *you* have to make a change is in your implementation of the `getRandomChar` method. In particular, you have to replace the call to the `Math.random` method with a call to the method `randomGenerator.nextDouble()`.

III.4 Usage

The program expects to get four command-line arguments and a text file, as follows:

% java LanguageModel *windowLength initialText textLength* fixed/random < *fileName*

For example, suppose we want to generate random text of length 1000 characters from the file `shakespeareinlove.text`, using a window length of 8, the initial text `FENNYMAN` (which happens to be the name of one of the characters in this play), and a random seed. To do so, we can execute the program as follows:

% java LanguageModel 8 "FENNYMAN" 1000 random < shakespeareinlove.text

(If you'll forget to include the redirection character "`<`", your program will do nothing, since the `StdIn.readChar` method will expect you to enter the training text from the keyboard).

If your code works well, the program will output a 1000 character long text that "looks like" the text listed in the given input file. Expect to see lots of spaces -- that's because the script of the play "Shakespeare in Love" contains many empty lines.

III.5 The Main method

So far in the course, when we wrote a class that generates objects, we normally wrote another class that tests it. This is not really necessary. Whenever you write a class, you can always include in it a static main method that tests it. This way the class will carry its own testing code, which is a good programming practice.

That's what we do in the `LanguageModel` class. This class includes several non-static methods, like `train` and `generate`, several private static "helper" method, and one public static `main` method, designed for testing purposes. The code of the `main` method is supplied, as follows:

```
/**
 * A Test of the LanguageModel class.
 * Learns a given corpus (text file) and generates text from it.
 */
public static void main(String []args) {
    int windowLength = Integer.parseInt(args[0]);  // window length
    String initialText = args[1];                  // initial text
    int textLength = Integer.parseInt(args[2]);    // size of generated text
    boolean random = (args[3].equals("random") ? true : false);  // random / fixed seed

    LanguageModel lm;

    // Creates a language model with the given window length and random/fixed seed
    if (random) {
        // the generate method will use a random seed
        lm = new LanguageModel(windowLength);
    } else {
        // the generate method will use a fixed seed = 20 (for testing purposes)
        lm = new LanguageModel(windowLength, 20);
    }

    // Trains the model, creating the map.
    lm.train();

    // Generates text, and prints it.
    System.out.println(lm.generate(initialText,textLength));
}
```

III.6 Things to do

Assuming that you went through all the steps described in Stages I and II, you are now ready to complete the project. Start by copy-pasting all the relevant methods that you wrote so far into the supplied `LanguageModel` class. Then change the `getRandomChar` method, as explained above (and re-test it). Finally, write the code of the `generate` method.

Test your code by running it on the two supplied files: `shakespeareinlove.text`, and `originofspecies.txt`. This can be done using program executions like:

% java LanguageModel 8 “FENNYMAN” 1000 random < shakespeareinlove.text

% java LanguageModel 8 “SELECTION” 1000 random < origin of species.text

The official test must be done as follows:

% java LanguageModel 6 “WESSEX” 60 fixed < shakespeareinlove.text

If your program is operating correctly, it will produce the following output, exactly:

```
WESSEX

  Elizabeth

      amazed. The hill
```

III.7 Submission

Assuming that you went through all the steps described in Stages I and II, you are now ready to complete the project. Start by copy-pasting (carefully) all the relevant methods that you wrote so far into the new supplied `LanguageModel.java` class. Then modify your `getRandomChar` method implementation, as explained above. Finally, write the code of the `generate` method.

Submission: Submit two files only: `CharProb.java`, and `LanguageModel.java`. There is no need to submit any testing code. We will do our own testing.

Deadline: Sunday, January 21, 2017, 23:55.