Homework 9: Machine Learning, Stage II

Like most machine learning (ML) algorithms, the algorithm that we will use for generating texts is based on two independent and well-defined modules: *training*, and *generation*. The training module creates a data structure that serves as a *language model*, and the generation module then uses this model for generating as many texts as we please, in any specified length.

The main input of the training stage is a large text file, from which the program "learns" typical textual patterns that characterize this particular text. This so-called "corpus" may be a novel, or a scientific book, or any other substantial body of text that allows the program to "learn" how to generate "similar" texts. As a rule, the larger the corpus, the better will be the training. We now turn to describe how we represent the language model learned by the program, and how the learning algorithm constructs this model.

Before we go on, a word of caution about the so-called learning process that we are about to describe. The curious reader may think that we are going to learn sophisticated linguistic aspects like sentence composition etc. In fact, we will learn nothing more than the sequential patterns in which *characters* are arranged in the given corpus. It is quite shocking and somewhat embarrassing that that's all that is needed for automatically synthesizing good looking texts.

II.1 Representation

For the sake of illustration, consider the following single sentence corpus, attributed to Galileo Galilei (we use underscores to mark space characters):

"you_cannot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself."

Let's focus on some two-character strings that appear in this corpus. Taking three arbitrary examples, we'll focus on "hi", "im", and "ms". In order to learn the textual patterns in this corpus, we wish to find which character appears just after each one of these two-character strings. So, if we'll process the corpus with this pattern in mind, we will find that the string "hi" is followed twice by the character 'n', and twice by 'm'. The string "im" is followed once by a space character, and once by 's'. Finally, the string "ms" appears only once, followed by 'e'.

The textual patterns observed above can be represented using the following data structures:

```
"hi": (('n', 2, 0.5, 0.5)('m', 2, 0.5, 1.0))   // The string "hi" is followed either by the character 'n',
                                                //  with probability 0.5, or by 'm', with probability 0.5

"im": (('_', 1, 0.5, 0.5)('s', 1, 0.5, 1))     // The string "im" is followed either by the space character,
                                                //  with probability 0.5, or by 's', with probability 0.5

"ms": (('e', 1, 1.0, 1.0))                      // The string "ms" is followed by 'e' with probability 1.0.
```

Each list element represents a character, followed by the number of times that it appears in the corpus just after the corresponding 2-letter string. Then come the corresponding probability and cumulative probability of observing this particular pattern in the corpus. Note that these two probabilities can be calculated directly from (i) the count value, and (ii) the total count values across the list.

How can we represent these data structures in Java? The list elements can be represented as `CharProb` objects, the lists themselves can be represented as `LinkedList<CharProb>` objects, and the collection of the (`String`, `LinkedList<CharProb>`) mappings can be represented as a `HashMap<String, LinkedList<CharProb>>` object. The complete data structure learned from the Galileo corpus is shown in appendix II-A. Take a look now.

II.2 Training

Our training strategy is based on moving a fixed-size "window" over the corpus, and recording which character appears just after this window. For example, suppose we use a 4-character window to learn the Galileo corpus. Here are the first three windows that the program will process:

"(you_)cannot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "you_"

"y(ou_c)annot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "ou_c"

"yo(u_ca)nnot_teach_a_man_anything;_you_can_only_help_him_find_it_within_himself." // window = "u_ca"

The complete training process can now be described as follows. The program proceeds only forward within the corpus, reading and processing one character at a time. The program starts by reading just enough characters to form the first window, which happens to be "you_". The program will then read the character 'c', and record in the map the fact that 'c' follows the window "you_". The program will then change the window to "ou_c", read the character 'a', and record in the map the fact that 'a' follows "ou_c". The program will then change the window to "u_ca", read the character 'n', and record in the map the fact that "n" follows "u_ca". And so on and so forth.

Note that when we say "the program records that the character appears just after the window", there are actually two very different cases to handle. If the window is seen for the first time (*the window is not in the map*), we add a new mapping to the map. If the window was already seen before (*the window is in the map*), we update its respective list. For example, when processing the above corpus, the window "you_" will appear twice, and each time it must be handled differently.

Appendix A shows the result of this learning process, when applied to the Galillo corpus with window size = 2.

II-3. Implementation

Appendix B proposes how to implement the logic just described, using pseudo code. Your job is to turn this into executable Java code, and test it on the Galileo corpus.

The training process described above is implemented by the `train` method of the supplied `LearningModel` class..

The code and methods that you developed in in Stage I of this project cannot be used directly in the implementation of Stage II. In other words, the `Tools` class plays no role in the training process (it will play a role in the text generation process). However, the programming experience that you've gained in Stage I will serve you well in Stage II.

The "map" which is mentioned in this document several times is implemented in this project as a HashMap object, which is an instance of Java's HashMap class.

How can you tell if a HashMap instance contains a certain key? Java's HashMap class uses the following convention. You can always call the method `get(`*key*`)`, using any key that comes to your mind. If there is such a key in the map, the method returns its corresponding value. If there is no such key in the map, the method returns `null`. The latter event can be used to conclude that the map does not contain the given key.

II.4 Usage

At this stage we focus on developing and testing the training process only. The `LanguageModel` class has a supplied `main` method that calls the `train` method and prints the resulting map. The corpus is the supplied `test.txt` file, which contains the one sentence Galileo quote that we used all along. We execute the program as follows:

```
% java LanguageModel 2 < test.txt
```

The single argument is the window size. The redirection causes the program to read its inputs from the supplied text file (in other words, standard input is *bound* to `test.txt`, or fto any other text file supplied by the user)

If your code works well, the output should produce the output listed in Appendix A.

The code of the `main` method is given, as follows:

```
// Learns the text that comes from standard input,
// using the window length given in args[0],
// and prints the resulting map.
public static void main(String[] args) {
   int windowLength = Integer.parseInt(args[0]);
   // Constructs a learning model
   LanguageModel lm = new LanguageModel(windowLength);
   // Builds the language model
   lm.train();
   // Prints the resulting map
   System.out.println(lm);
}
```

II-A. Appendix: Language Model Example

Suppose that we execute the `train` method of the `LanguageModel` class on the corpus "you cannot teach a man anything; you can only help him find it within himself.", using a 2-character window. As it turns out, this 78 character-long string has 54 unique 2-character sequences like "hi", "lp", etc. Below is the language model that will be constructed by the `train` method.

```
hi : ((n 2 0.5 0.5)(m 2 0.5 1.0))
lp : ((  1 1.0 1.0))
ly : ((  1 1.0 1.0))
ma : ((n 1 1.0 1.0))
yo : ((u 2 1.0 1.0))
yt : ((h 1 1.0 1.0))
ea : ((c 1 1.0 1.0))
ac : ((h 1 1.0 1.0))
im : ((  1 0.5 0.5)(s 1 0.5 1.0))
in : ((g 1 0.33 0.33)(d 1 0.33 0.66)(  1 0.33 1.0))
ms : ((e 1 1.0 1.0))
el : ((p 1 0.5 0.5)(f 1 0.5 1.0))
it : ((  1 0.5 0.5)(h 1 0.5 1.0))
t  : ((t 1 0.5 0.5)(w 1 0.5 1.0))
an : ((n 1 0.25 0.25)(  2 0.5 0.75), (y 1 0.25 1.0))
p  : ((h 1 1.0 1.0))
g; : ((  1 1.0 1.0))
nd : ((  1 1.0 1.0))
h  : ((a 1 1.0 1.0))
ng : ((; 1 1.0 1.0))
d  : ((i 1 1.0 1.0))
nl : ((y 1 1.0 1.0))
nn : ((o 1 1.0 1.0))
no : ((t 1 1.0 1.0))
 a : ((  1 0.5 0.5)(n 1 0.5 1.0))
 c : ((a 2 1.0 1.0))
fi : ((n 1 1.0 1.0))
;  : ((y 1 1.0 1.0))
 f : ((i 1 1.0 1.0))
y  : ((h 1 1.0 1.0))
 h : ((e 1 0.33 0.33)(i 2 0.66 1.0))
 i : ((t 1 1.0 1.0))
u  : ((c 2 1.0 1.0))
ny : ((t 1 1.0 1.0))
 m : ((a 1 1.0 1.0))
 o : ((n 1 1.0 1.0))
wi : ((t 1 1.0 1.0))
se : ((l 1 1.0 1.0))
m  : ((f 1 1.0 1.0))
 t : ((e 1 1.0 1.0))
 w : ((i 1 1.0 1.0))
 y : ((o 1 1.0 1.0))
ca : ((n 2 1.0 1.0))
a  : ((m 1 1.0 1.0))
on : ((l 1 1.0 1.0))
ot : ((  1 1.0 1.0))
ch : ((  1 1.0 1.0))
ou : ((  2 1.0 1.0))
te : ((a 1 1.0 1.0))
n  : ((a 1 0.33 0.33)(o 1 0.33 0.66)(h 1 0.33 1.0))
th : ((i 2 1.0 1.0))
lf : ((. 1 1.0 1.0))
he : ((l 1 1.0 1.0))
```

(The above was printed by the `toString` method of `LanguageModel`. The order of the mappings is arbitrary, since the collection is a map. Each list element shows the values of the four `CharProb` fields: `chr`, `count`, `p`, and `cp`.)

II-B. Code

The following (quite detailed) pseudo code can be used for implementing the train method. Replace the blue lines with your own Java code.

```
public void train() {
   String window = "";
   char c;

   // Reads just enough characters to form the first window
   code: Performs the action described above.

   // Processes the entire text, one character at a time
   while (!StdIn.isEmpty()) {
      // Gets the next character
      c  = StdIn.readChar();
      // Checks if the window is already in the map
      code: tries to get the list of this window from the map.
      Let’s call the retrieved list “probs” (it may be null)
      // If the window was not found in the map
      code: the if statement described above {
         // Creates a new empty list, and adds (window,list) to the map
         code: Performs the action described above.
         Let’s call the newly created list “probs”
      }
      // Calculates the counts of the current character.
      calculateCounts(probs, c);

       // Advances the window: adds c to the window’s end, and deletes the
       // window's first character.
       code: Performs the action described above.
   }

   // The entire file has been processed, and all the characters have been counted.
   // Proceeds to compute and set the p and cp fields of all the CharProb objects
   // in each linked list in the map.
   for (LinkedList<CharProb> probs : probabilities.values())
      calculateProbabilities(probs);
   }

   // If the given character is found in the given list, increments its count;
   // Otherwise, constructs a new CharProb object and adds it to the given list.
   private void calculateCounts(LinkedList<CharProb> probs, char c) {
      Put your code here
   }

   // Calculates and sets the probabilities (p and cp fields) of all the
   // characters in the given list.
   private void calculateProbabilities(LinkedList<CharProb> probs) {
      Put your code here.
      Tip: requires going through the list twice.
   }
}
```

What to Do

Complete the implementation of the given `LanguageModel` class, and test it using the supplied `test.txt` file with window length = 2.

There is no need to submit anything. The last and final stage of this project, along with submission guidelines, will be published in a day or two.

For your own progress and sanity, you must complete this part of the project no later than Monday, 23:55.