Homework 10: Text Generation / Stage II

Below are two dialogues, taken from the play *Romeo and Juliet*. One was written by William Shakespeare, and one by the program that we are writing in this project. Can you tell which dialogue was written by William, and which by Java?

JULIET. Farewell, farewell; good night, all this night I hold it not then? ROMEO. This window falls, Like death to any he that shot so trim When he bestrides the law, And turns it to exile; there art thou fishified! Now is he for the third in your bosom: the very pin of his head. Go hence, for I am coming.	JULIET. What man art thou that, thus bescreen'd in night, so stumblest on my counsel? ROMEO. By a name I know not how to tell thee who I am: My name, dear saint, is hateful to myself, Because it is an enemy to thee. Had I it written, I would tear the word.

Well, the left dialogue was written by Java, and the right by William.

Quite striking, isn't it? In Stage I of this project you built the basic infrastructure needed for performing this hocus focus. In Stage II you will implement a *training algorithm*, designed to learn a given body of text, like a play written by William Shakespeare. In Stage III you will implement a *text generation algorithm* that uses the learned knowledge for creating all the random plays that William Shakespeare did not manage to write in his career, which was unfortunately finite.

II.1 Corpus

Following the terminology of Computational Linguistics and Natural Language Processing (NLP), we will use the term *corpus* to refer to the "training text" that our program will learn. We wish to analyze this corpus in order to develop the ability to create texts that are similar to that corpus. Normally, the corpus is a huge textual resource, like all the books written by a certain author. When we say "all the books" we mean "all the books that we can get our hands on". Although most digital books can be obtained only by buying them, there are many classics whose copyrights have expired, like all the books that were published in the USA before 1923, and many newer books whose authors decided to put them in the public domain, like this excellent book.

For example, the Gutenberg Project is a library of 60,000+ freely available eBooks. Each eBook can be downloaded from the Gutenberg website in several different formats, designed for different reading platforms (like Kindle). One of these formats is a simple text file containing a long sequence of characters – exactly what we need for the training purposes of this project. In particular, we supply the following corpuses (the first two were downloaded from Gutenberg):

- `hemm.txt`: A collection of short stories, written by Ernest Hemingway
- `shake.txt`: The play *Romeo and Juliet*, written by William Shakespeare
- `word.txt`: A single word, for testing purposes
- `sentence.txt`: A single sentence, for testing purposes

These files can be inspected using a simple text editor. Java programs can read them using the supplied `StdIn.java` class.

II.2 Representation

Technically speaking, a corpus is a sequence of characters, stored in a text file. For example, consider the following single sentence, attributed to Galileo Galilei, and stored in the file `sentence.txt`:

```
"you cannot teach a man anything; you can only help him find it within himself."
```

To illustrate some of the concepts that we will use in our text analysis, let's focus on two-character strings that appear in this corpus. Taking an arbitrary example, consider "hi". This string is followed twice by the character 'n', and twice by 'm'. Taking another arbitrary example, the string "im" is followed once by a space character, and once by 's'. One more example: the string "ms" appears only once, followed by 'e'. For reasons that will become clear soon, the two-character strings described above are called "windows". The textual patterns that we found about these windows can be represented as follows:

"hi": (('n', 2, 0, 0) ('m', 2, 0, 0)) // The window "hi" is followed by 'n', twice, and by 'm', twice

"im": ((' ', 1, 0, 0) ('s', 1, 0, 0)) // The window "im" is followed by ' ' (space), once, and by 's', once

"ms": (('e', 1, 0, 0)) // The window "ms" is followed by 'e', once.

Each window (two-character string) is associated with a list. Each element in the list records a character and the number of times that the character appears in the corpus just after the corresponding window. If we'll calculate the probabilities of these character occurrences, as we did in Stage I of this project, we will get the following data:

"hi": (('n', 2, 0.5, 0.5) ('m', 2, 0.5, 1.0)) // The window "hi" is followed by 'n' with probability 0.5,
// and by 'm' with probability 0.5

"im": ((' ', 1, 0.5, 0.5) ('s', 1, 0.5, 1)) // The window "im" is followed by a space with probability 0.5,
// and by 's' with probability 0.5

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

We call these structured relations *mappings*: Each relation consists of a `String` object (the window) which is mapped on a `List` object (a sequence of `CharData` objects). Suppose that we have a huge collection of many thousands of such mappings. How can we store and use this collection? Java provides a perfect solution for this need: the generic class `HashMap<K,V>`, in which `K` stands for *key* and `V` stands for *value*. In our application, `K` will be a `String` object, and `V` will be a `List` object.

Such a data structure can be constructed using the Java statement:

```
HashMap<String, List> map = new HashMap<String, List>();
```

The left side declares a variable named `map` of type `HashMap<String, List>`, and the right side calls a `HashMap` constructor. The constructor creates an empty map whose base address is then assigned to the `map` variable. Following this construction, we can add mappings to this map using the method call `map.put(str, list)`, where `str` is a `String` object and `list` is a `List` object (a sequence of `CharData` objects). Likewise, we can get the specific list associated with the string `str` using the method call `map.get(str)`. If `str` is in the map, the method will return its value, which is a `List` object. Otherwise, the method will return `null`.

Like all abstractions, the beauty of `HashMap` is that we don't have to worry about its implementation. We can happily *put* millions of `<str, list>` mappings into the map, whenever the need arises, and we can get the specific `list` of any given `str`, *instantaneously*, whenever the need arises. The remarkable *map* data structure and the hashing algorithm that enables its implementation were described in lecture 10-2.

II.3 Training

The training process starts with a given corpus and window size, and ends with a map containing the mappings described above. The 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 the single sentence corpus stored in `sentence.txt` and a window of size 4. The training process will move the window through the text from left to right, one character at a time, as follows:

```
(you )cannot teach a man anything; you can only help him find it within himself.
y(ou c)annot teach a man anything; you can only help him find it within himself.
yo(u ca)nnot teach a man anything; you can only help him find it within himself.
you( can)not teach a man anything; you can only help him find it within himself.
you (cann)ot teach a man anything; you can only help him find it within himself.
...
you cannot teach a man anything; you can only help him find it within hi(msel)f.
you cannot teach a man anything; you can only help him find it within him(self).
you cannot teach a man anything; you can only help him find it within hims(elf.)
```

The complete training process can now be described as follows. The program starts by reading just enough characters from the corpus to create the first window, which, in this case, happens to be the 4-character string "`you `". Next, the program reads the next character in the corpus, which happens to be '`c`', and records in the map that "`you `" is followed by '`c`'. In the next iteration the program moves the window one character forward, resulting with the new window "`ou c`". The program then reads the next character, which is '`a`', and records in the map that "`ou c`" is followed by '`a`'. In the next iteration the program moves the window forward to "`u ca`", reads the next character '`n`', and records in the map that "`uc a`" is followed by '`n`'. And so on and so forth. In the last iteration the program records in the map that "`self`" is followed by '`.`'.

Note that when we say "the program records that the window is followed by a character", there are actually three 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 "`thin`" will appear twice. In the first time, the program will add the following mapping to the map

"thin": ((‘g‘, 1, 0, 0))

In the second time that "`thin`" is found in the corpus, the program will update the mapping thus:

"thin": ((‘g‘, 1, 0, 0) (‘ ‘, 1, 0, 0))

If "`thin`" were to appear again in the corpus, followed again by the character '`g`', the program would update the mapping thus:

"thin": ((‘g‘, 2, 0, 0) (‘ ‘, 1, 0, 0))

Appendix II.6 shows the result of this learning process, when applied to the `sentence.txt` corpus with window size = 2.

II.4 TextTrain

We will now describe a program that implements the training process described above. The program takes a given text file and window size as inputs, and produces a map as output.

Usage: `% java TextTrain` *windowLength fileName*

Example: `% java TextTrain 2 sentence.txt`

Appendix II.6 shows the result of this program execution. If you haven't done it so far, inspect this appendix now.

II.5 Task II

(Task I is what you did in Stage I of this project). Implement and test the `TextTrain` program.

Implementation Notes

Class structure

The `TextTrain` class is a collection of static methods, also known as *functions*. The program represents the *map* and the *window size* as static variables that all the functions in the class can access. Read the given class declaration and identify these class-level variables.

Main:

The `main` function starts by getting the command-line arguments and calling the `init()` function. It then calls the `train()` function, and finally prints the map created by the training process. The code of the `main` function is given.

Init:

In general, "init" functions are used to initialize and set up things. In that respect, they are similar to constructors, although they create no objects. They are sometimes used in classes that are not object-oriented, like the `TextTrain` class. Read the code of the `init()` function, which is given, and make sure that you understand what it is doing.

Train:

This function implements the training process described in section II.3. Note that the training process consists of two main stages. First, the `train()` function creates a map in which the `p` and `pp` fields of each `CharData` object in each list are not yet calculated. Tip: the `List` method `update` plays a big part in this construction. Second, the `train()` function calls the `calculateProbabilities()` method on each list in the map. This code is given, and we hope that you will not miss its beauty. Explanation: the `values()` method, which is part of the `HashMap` class API, returns a *collection* of all the *value* objects in the map. This is an *iterable collection*, meaning that you can use it in for-each loops, as we do here.

CalculatePobabilities:

This function was implemented and tested in Stage I of this project. Assuming that you wrote it already, copy-paste your code from the `test2.java` class described in Stage I.

MapString:

This function creates outputs like the one printed in Appendix II.6. The function iterates through all the *keys* in the map. For each key, the function prints the key itself (the window) and the *value* associated with that key (a `List` object). The `toString` method of the `List` class will come very handy here. Explanation: the `keySet()` method, which is part of the `HashMap` class API, returns a collection of all the *key* objects in the map (in this application – all the windows). This is an *iterable collection*, meaning that you can use it in for-each loops, as we do here.

II.6 Appendix: Map Examples

The following output illustrates the application of the `TextTrain` program to the file `word.txt`, using a window of length 2. The file contains the corpus:

```
committee
```

```
% java TextTrain 2 test1.txt
```

```
mm: ((i 1 1.0 1.0))
tt: ((e 1 1.0 1.0))
te: ((e 1 1.0 1.0))
it: ((t 1 1.0 1.0))
co: ((m 1 1.0 1.0))
mi: ((t 1 1.0 1.0))
om: ((m 1 1.0 1.0))
```

The following output illustrates the application of the `TextTrain` program to the file `sentence.txt`, using a window of length 2. The file contains the corpus:

```
you cannot teach a man anything; you can only help him find it within himself.
```

% java TextTrain 2 sentence.txt

```
hi: ((m 2 0.5 0.5), (n 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: ((s 1 0.5 0.5), (  1 0.5 1.0))
in: ((  1 0.3333333333333333 0.3333333333333333), (d 1 0.3333333333333333 0.6666666666666666), (g 1
0.3333333333333333 1.0))
ms: ((e 1 1.0 1.0))
el: ((f 1 0.5 0.5), (p 1 0.5 1.0))
it: ((h 1 0.5 0.5), (  1 0.5 1.0))
t : ((w 1 0.5 0.5), (t 1 0.5 1.0))
an: ((y 1 0.25 0.25), (  2 0.5 0.75), (n 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: ((n 1 0.5 0.5), (  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: ((i 2 0.6666666666666666 0.6666666666666666), (e 1 0.3333333333333333 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 : ((h 1 0.3333333333333333 0.3333333333333333), (o 1 0.3333333333333333 0.6666666666666666), (a 1
0.3333333333333333 1.0))
th: ((i 2 1.0 1.0))
lf: ((. 1 1.0 1.0))
he: ((l 1 1.0 1.0))
```

Things to observe:

- Each window appears in the map exactly once, even if it appears in the corpus more than once.
- In every list of some window, each character appears at most once, even if it follows the window in the corpus more than once. The number of times that the character follows the window is recorded in its `count` field.
- The order of the mappings is insignificant, and serves no purpose in this application. The mappings are stored in a *map*, which is an unordered collection (like a set).
- The order of the elements in the list associated with each window is also insignificant. The `List` implementation that we use in this project adds new elements to the beginning of the list. However, the random text generation process described in Stage III does not care in which order the lists are managed.

TO DO

Create a folder for Stage II, and put in it the following (fully developed) classes:

- `Node.java` (from Stage I)
- `CharData.java` (from Stage I)
- `List.java` (from Stage I)
- `StdIn.java` (given)
- `TextTrain.java` (which you have to complete in Stage II)

Stage III, which is relatively small, will be published on Tuesday. The submission deadline of the entire project (stages I, II, and III) is January 22, 23:55.