CS2020: Data Structures and Algorithms (Accelerated)

Automatic Writing

Problem Set 6 *Due: Sunday April 5, 11:59pm*

Overview

Writing-intensive modules can be hard: so many 10-page essays, and not nearly enough time to catch up on the latest movies. CS2020 is here to help. For this problem set, you will develop an automatic writing program that can easily produce pages and pages of new text. And it will adapt to your chosen style. If you use an old essay as input, your new essay will sounds just like it was written by you, If you use Shakespeare as input, your new essay will sounds as if it was written by the Bard.

The basic idea is to take an input text and calculate its statistical properties. For example, given a specific string "prope", what is the probability that the next letter is an 'r'? What is the probability that the next letter is a 'q'? Your program will take a text as input, calculate this statistical information, and then use it to produce a reasonable output text.

Claude Shannon first suggested this technique in a seminal paper *A Mathematical Theory of Communication* (1948). This paper contained many revolutionary ideas, but one of them was to use a *Markov Chain* to model the statistical properties of a particular text. Markov Chains are now used everywhere; for example, the Google PageRank algorithm is built on ideas related to Markov Chains.

Markov Models

A Markov Model captures the probability of a specific letter appearing after a specific preceding string. The *order* of the Markov model is the length of that string. For example, a Markov model of order 1 might contain the following:

a	b	1/2
a	c	1/2
b	d	1
c	a	1/3
c	b	1/3
c	d	1/3
d	a	1

This implies that after the letter 'a', you find a 'b' with probability 1/2 and a 'c' with probability 1/2. After the letter 'b' you always find a 'd'. After the letter 'c', you find each other letter 'a', 'b', or 'd' with probability 1/3. After letter 'd', you always find an 'a'. The following text would generate this Markov model:

```
a b d a c a b d a c b d a b d a c d a
```

Notice that in this text, the character 'a' is followed by a 'b' three times, and 'a' is followed by a 'c' three times. Similarly, 'b' is always followed by a 'd', and 'd' is always followed by an 'a'. The character 'c' is followed by an 'a' once, a 'b' once, and a 'd' once.

A Markov model of order 2 records how likely a given character is to follow a string of length 2. Here is an example of a Markov model of order 2:

ab	c	1/2
ab	d	1/2
bc	d	1
bd	d	1
cd	a	1/2
cd	d	1/2
da	b	1
dd	a	1

The following text would generate this Markov model:

```
a b c d a b d d a b c d d a b d
```

Notice that after the string 'ab', the letter 'c' appears twice and the letter 'd' appears twice. After the string 'bc', you always get the letter 'd', and after the string 'bd', you always get the letter 'd'. Et cetera.

Problem Details

For this problem, you will submit two Java classes: `MarkovModel` and `TextGenerator`.

Markov Model. The `MarkovModel` class should implement the following methods:

> `MarkovModel(String text, int order)`: creates a Markov Model from the specified text of the specified order. You can assume that the order will be at least 1.
>
> `int order()`: Returns the order of the Markov model.
>
> `int getFrequency(String kgram)`: Returns the number of times the specified string `kgram` appears in the input text. The `kgram` must be the length specified by the order of the Markov model.
>
> `int getFrequency(String kgram, char c)`: Returns the number of times the specified character 'c' appears immediately after the string `kgram` in the input text. The `kgram` must be the length specified by the order of the Markov model.
>
> `char nextCharacter(String kgram)`: Returns a random character. The probability of a character 'c' should be equal to `getFrequency(kgram,c)/getFrequency(kgram)`. That is, the probability of character 'c' should be equal to the frequency that 'c' follows the string `kgram` in the text. If there is *no possible* next character, then return an indicator of such, e.g.: `final char NOCHARACTER = (char)(255)`. The `kgram` must be the length specified by the order of the Markov model.

		frequency			probability		
		a	c	g	a	c	g
aa	1	1	0	0	1	0	0
ag	4	2	0	2	2/4	0	2/4
cg	1	1	0	0	1	0	0
ga	5	1	0	4	1/5	0	4/5
gc	1	0	0	1	0	0	1
gg	3	1	1	1	1/3	1/3	1/3

Figure 1: Markov Model produced by the string `gagggagaggcgagaaa`.

`void setRandomSeed(long s)`: Sets the random seed to be used by the random number generator. If you set the same seed, it will produce the same sequence of random numbers, and hence the same sequence of characters. We will use this only in testing your program.

Your program should throw an exception if there is any error. For example, if the input `kgram` is not of the right length (i.e., is not of length `order()`), then it should throw an exception indicating an invalid input parameter. You may assume that every character in the text is a standard ASCII character, i.e., each `char` is between 0 and 127. Figure 1 has an example of the information stored by your Markov Model for a specific example string.

There are several approaches to designing the `MarkovModel` class. One approach is as follows: Use a symbol table (i.e., a hash table) that maps strings of length k (where k is the order of the Markov model) to an array containing 127 integers. The array records the number of time each character follows the given string. For example, the character 'a' is 97, in ASCII. Hence, if $k = 2$, given an input string 'xya', you would add to your hash table an entry with the key equal to 'xy' and the value equal to an array of integers where `value[97]=1`. You may also use an alternate solution, as long as it efficiently supports the required operations.

Text Generator. The text generator class takes three input parameters (i.e., the main method has argument `arg[0], arg[1], arg[2]`):

- k, the order of the Markov model;
- n, the number of characters to generate;
- the filename of the text to use as a model.

Your program should read in the textfile as a string, and create a Markov Model of order k from the input text. Finally, it should generate text from the Markov model.

In order to generate text, begin with a `String kgram` equal to the first k letters from your text file. Generate the next character by calling `nextCharacter` on your Markov Model, using the initial `kgram` as your input string. Then update the `kgram`, dropping the first character and adding the newly generated character to the end. Finally, output the new character and continue until you have generated n characters. If you ever reach a point where there is *no possible* next character, then begin again, initializing the `kgram` to the first k characters in the text file.

For example, using the Markov Model in Figure 1, the `kgram` is intiially `ga`. If the Markov Model outputs a 'g' for the `nextCharacter`, then update the `kgram` to *ag*. If it outputs 'a' for the `nextCharacter`, then output the `kgram` to *aa*.

Optional Experiments

You may want to experiment with different values of k to determine which values yield reasonable sounding texts. If k is too large, then the text output will be identical to the input text. On the other hand, if k is too small, then the output text is quite garbled and ungrammatical English. There is a small range of k for which the output text both sounds like real English, but also sounds like a new and unique text.

Bonus Extensions

You might also experiment with using words instead of characters. For example, instead of looking at the probability that character 'c' comes after the string 'cra', you could look at the probability that the word *cat* comes after the word *yellow* (i.e., order 1), or the probability that *cat* comes after the phrase *the vicious yellow*. If you develop your Markov Model based on words, you might get a more interesting text, as long as you begin with a sufficiently long text.

In order to use words instead of characters, you will have to design your `MarkovModel` class more carefully, as the solution described in this problem set will not work.

Optional Competition

Submit the best, most interesting text that your program produces. Post your best, most creative work to the CS2020 Facebook group (`facebook.com/groups/cs2020.2014/`). The one(s) with the most *like*s win(s)! (The winner(s) will get a small bonus.) You could aim to fit several criteria: (i) plausibility (i.e., does it read like a real text), (ii) novelty, and (iii) humor.

In order to generate an interesting text, you will need to find a good (long) source text. In the past, people have used legal documents (e.g., the text of a specific law), Shakespeare, news reports, and/or arbitrary texts from Project Gutenberg (`gutenberg.org`). You may splice together 2-3 input texts. You may also experiment with modifying the initial `kgram`. Instead of using the first k letters of the text as an initial `kgram`, you may choose an alternate choice of k characters.

On a closing note:

> *"Half the fire a funny little man began to stay at heavens. 'How beautiful this kingdom' said their new emperor."*