CS 251 Intermediate Programming
Lab 6: Markov Chain Text Generation
Week of April 6–11, 2020.

originally by Brooke Chenoweth, modified by Aayush Gupta and Tom Hayes

HAMLET, PRINCE OF DENMARK

by William Shakespeare

PERSONS REPRESENTED.

Claudius, King of Denmark.
Hamlet, Son to the three men at her reverend helm, she sported there a tiller; and that tiller in a ring; and so closely shoulder to shoulder, that a Titanic circus-rider might easily have over-arched the middle when midway, and comes to him, and always at hand to them, they all dispersed; and Ahab retired within his cabin.

CHAPTER 22. Merry Christmas.

At length, towards noon, upon the final consequence. Why so? Because a laugh's the wisest, easiest answer to any utterance of harmony; I have not yet arrived; and there these silent islands of Sumatra and Java, freighted with the lightning, that showed the uneasy, if not painful, consciousness of being dragged along the undulating ridges. But you must be the case, and occasionally with a dull, spotted, green mould, in consequence of being kept in a life which, to your now equally abhorred and abhorring, landed world, is more oblivious than death. Come hither! put up THY gravestone, too, within the churchyard, and come hither, till we marry thee!"

Hearkening to these delights.

Ros. We shall, my lord.

[Exeunt Rosencrantz and Guildenstern.]

The previous gibberish text was generated by a word-based order 2 Markov Chain that had been given the texts of *Hamlet* and *Moby Dick*. (We will be discussing Markov Chains in lecture this week, so you'll learn what that means.)

1 Markov Chains

A Markov Chain is a mathematical model describing a sequence of objects called *states*, in which each state is randomly chosen in a way that depends only on the immediately previous state.

1.1 Markov Random Text Generator

A Markov random text generator, which we will abbreviate MRTG, is a special kind of Markov chain that outputs a string of ”text” that looks, locally, like human-generated text. However, as you look at bigger pieces of it, it looks more and more like gibberish. This is one of the easiest and least sophisticated means of generating realistic-looking text. So, how does a MRTG work?

To set it up, you will need a *corpus* of text that the MRTG will mimic. For example, this can be the text of one or more novels or plays.

Next, you need to build up a table of n-grams, which are just n-tuples of consecutive words from the text. These are the *states* of your Markov chain. Here, n is a positive integer parameter, called the *order* of your MRTG. For each n-gram, you use your corpus to build up a table of *transitions* to a possible next n-gram. A little example should make this more clear.

For example, suppose we build up a first-order MRTG from the ridiculously tiny corpus, ”Mary had a little lamb, little lamb, little lamb. Mary had a little lamb whose fleece was white as snow. Everywhere that Mary went, Mary went, Mary went, everywhere that Mary went, the lamb was sure to go.”

Now, since our $n = 1$, our states are single words from this corpus. For now, let's just strip out and discard all the punctuation and capitalization. What are the transitions from the state ”mary”? Well, ”mary” occurs six times in the corpus. Four times, ”mary” is followed by ”went”, and twice by ”had”. So our MRTG will mimic this by choosing the transition from ”mary” to ”went” four times out of six, and ”mary” to ”had” two times out of six. Similarly, ”fleece” is always followed by ”was” in the corpus, so the MRTG will do the same. ”was” is followed by ”white” once and by ”sure” once, so the MRTG will choose the corresponding two transitions with equal probability, whenever its state is ”was”.

As a second example, suppose we had a bigram (order 2) model from the same corpus. In this case, our states are pairs of words that appear consecutively in the text. Then, for example, from the state ”mary went”, which appears four times in the corpus, we have two transitions to ”went mary”, one to ”went everywhere”, and one to ”went the”. The MRTG will mimic these probabilities. Note that the transitions here slide the first word out of the bigram, and slide in a new second word, so that the two states always overlap in one word. For an order n MRTG, the transition will again discard only the first word from the previous state, sliding a single word in at the end to create the next state; thus consecutive states always overlap in $(n - 1)$ words.

1.2 Punctuation in the text

For the punctuation in the text, you have number of choices :

1. Avoid the punctuation
2. Consider the punctuation and the word as a single entity.
3. Consider the punctuation separate from the word.

Please start out by just stripping out and discarding all punctuation (and capitalization). Once you have a working MRTG, you can experiment with the other two options, if you like.

2 Math.random()

The built-in method Math.random() returns a pseudo-random double type number greater than or equal to 0.0 and less than 1.0.

Algorithm : To print 5 random numbers between 1 to 10

```
max = 10;
min = 1;
range = max - min + 1;
for i from 1 to 5 :
    rand = (Math.random() * range) + min;
    print(rand);

Sample Output :
5
3
2
8
2
```

3 java.util.HashMap

Last week, we talked in lecture about Maps and HashMaps. This is a good opportunity to use them. Your transition table should basically be a Map that has each n-gram from the text as a key, with the corresponding transition rule as its value.

I recommend defining your own classes for n-grams and for transition rules, as these are both natural classes of objects for this project.

Once you have done this, your transition table will be a variable of type `HashMap<NGram,TransitionRule>`.

4 Program Description

For this project, you will create a program that takes two integer parameters, `n` and `len`, and the name of a text file. Your program will create an n-gram MRTG based on the text in the file, and then output `len` words of gibberish in the style of the given corpus.

I expect you to submit at least three files:

- `NGram.java`, to define a class of MRTG states.
- `TransitionRule.java`. Each object in this class should specify a list of possible NGrams to transition to, along with their associated probabilities. It should have an `sample` method that generates a random transition according to the rule.
- `MarkovTextGenerator.java`, to contain the main method for the program.

Usage example:

```
> java MarkovTextGenerator 2 100 jabberwocky.txt
```

should generate 100 words of gibberish using an order-2 MRTG based on the given corpus. The generated text should be printed to standard output.

5 Important Hints and Guidelines

- To parse the corpus, use either a `BufferedReader` or `Scanner` to read in the text. The `String.split()` method is quite useful for splitting the lines up into words.
- There are advantages to making sure your corpus ends with an n-gram that occurs at least once earlier in the corpus. In this case, you will never run out of transitions. In case your corpus does not do this, you should add a special `TransitionRule` for dealing with n-grams that don't have a successor, for instance, start fresh with a randomly chosen n-gram and continue generating text.

6 Challenges

Want more Markov chain fun? Try some of these ideas, but make sure you don't break the original requirements.

- How else might you break up a text? Add additional options beyond words separated by non-alphabetical characters. You'll probably want to look into the `Pattern` class and regular expressions.
- How fast is your program? Do you have to wait when adding several novels? Try to track down where the time is going and improve it.

- Could you make a generic `MarkovChain<T>` class that can make chains of anything, not just `String`s? Your `MarkovTextGenerator` class could then simply extend `MarkovChain<NGram>` and do very little work itself.
- Have you generated some particularly amusing text? Share them in the discussion forum. What source files did you use and what settings?

7 Turning in your assignment

Submit your source files and the output of two sample runs using different corpuses and orders, and showing the command line arguments used to run the program. If you have done anything beyond the minimum requirements, please also include a readme file describing what you did in some detail.

Upload all of the above on Learn.