About Lab 7

In Lab 7 we will use maps -- both HashMaps and TreeMaps, to implement a Markov Model for generating text.

Markov models make the assumption that the future is determined by the present, not the past.

To be a little less philosophical, in a system that has a sequence of states, the next state is determined by the current state; we don't need to keep track of how we got to the current state. Modelers like this assumption because it makes models simpler.

Many Markov models go through a training phase where they can see the rewards and penalties of taking a particular action in a given state.

For text generation a Markov model has an integer parameter K, usually a small integer in the range from 2 to 10 or so. We model the "state" of the text generation by a string with the last K characters we have generated. For example, with the string "Hi, Mom!", if K is 2 the successive states while we progress through the string are "Hi", "i,", ", ", " M", "Mo", "om", and "m!".

In the training phase we analyze a training text to find all of its states and what the next character was for each instance of the state. In the generation phase we generate followup characters for the state with the same frequency as they occurred in the training text.

Here is a very simple example. Our training text is the string "ababbbabab" and we will use the Markov constant K=2. The training text has only 3 distinct states: "ab", "ba", and "bb".

- State "ab" is followed by 'a' twice and 'b' once.
- State "ba" is always followed by 'b'.
- State "bb" is followed by 'b' once and 'a' once.

Suppose we start generating text with "ab". That puts us in state "ab". We could follow this state with either an 'a' or a 'b'. We will choose randomly between these, making sure 'a' is twice as likely as 'b'. Suppose we choose 'b'. That makes our generated text "abb" and our new state (the last K characters generated) is "bb". From there 'a' and 'b' are equally likely; perhaps we choose 'a'. This gives us text "abba" and new state "ba". From there we must choose 'b', so our new text is "abbab' and our new state is "ab". This process can continue as long as you like.

In Lab 7 you will implement this. The program consists of

a) Class State , which has a string and a TreeMap<Character, Integer> whose keys are all of the followup characters for the state's string. In the previous example the State for string "ab" has 'a' associated with 2 and 'b' associated with 1.

Class State has methods

void add(char c) to note that we have just seen c as a followup to the state's string, and

char generate( ) that uses the state information to generate the next character. In the example we just saw for state "ab" generate( ) will return 'a' two-thirds of the time and 'b' one-third of the time.

b) Class MarkovModel, which has a HashMap<String, State> and two methods: one for training the model and one for generating text.

The training method is given a text file. It walks through the file finding all of its states and their followup characters, adding that information to the HashMap.

The generateText( ) method starts with an initial string, and goes into a loop. Each time around the loop it gets the current State, calls that State's generate( ) method to get the next character, then uses that character to switch to a new State.

c) Finally, class TextGenerator just directs traffic. It makes a new instance of class MarkovModel, and trains it on a file. It then calls the model's generateText( ) method and prints whatever this method returns.

All of the interesting action happens in the MarkovModel class.

In this lab we need to read a file character-by-character. Scanners aren't very good for this. Instead of Scanners we will use FileReaders. The FileReader constructor needs a file name:

```
FileReader fr = new FileReader("sample.txt");
```

We will use only one method of class FileReader: read( ). The read( ) method returns an int. If it is at the end of the file it returns -1. If it is not yet at the end of the file it returns a Unicode value that can be cast into a char.

For example, here is a method that prints a file to the console:

```
void printFile( String fName ) {
        try {
                FileReader fr = new FileReader( fName );
                Boolean done = false;
                while (!done ) {
                        int c = fr.read( );
                        if (c == -1)
                                done = true;
                        else System.out.print( (char) c );
                }
        }
        catch (FileNotFoundException e) {System.out.println( "Bad File");}
        catch (IOException e)  {System.out.println( "IO Error" );}
}
```

The FileReader constructor throws a FileNotFoundException, and the FileReader read( ) method throws an IOException, and loops like this need two catch clauses.

Note that FileReaders treat the newline character ‘\n’ just like any other character. The Markov model does as well. You don’t need to do anything in your model to generate line breaks; a line will end whenever your generate() method generates a ‘\n’ character.

Here is the order in which you will implement things:

A. The add() method for class State.
B. Class MarkovModel's train(String filename) method.
C. We give you a test program that trains a model on the string "agggcagcggggcg" and then prints all of the states of the model.
D. Class State's generate() method.
E. Class MarkovModel's generateText() method.
F. We give you an application program TextGenerator that takes a file name and trains a model on it, then uses this model to generate text.

Question: Why do you think we use a TreeMap in class State to store the information for a single state (all of the followup characters for the states' strings) and a HashMap in class MarkovModel to associate a state with a give string??