Caleb Allen
CST 338
17 June 2017

Markov Generator

Introduction:

This week you will create a “Markov Machine” that uses Markov Chains to simulate patterns.

Background Information:

What are Markov Chains?

Before we begin, please watch the following YouTube videos from Khan Academy; which will explain what Markov Chains are and illustrate how they can be used to simulate language.

https://www.youtube.com/watch?v=Ws63I3F7Moc

https://www.youtube.com/watch?v=WyAtOqfCiBw

We will be using Markov Chains to generate names for fantasy characters in the same way this site does: http://fantasygen.herokuapp.com/#/markovChain

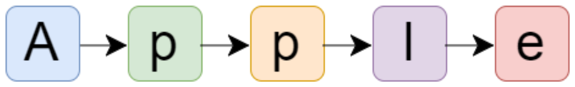
Markov Chains will mimic the patterns of whatever names we input and create new sequences that “fit in” with the originals.

How can we create them?

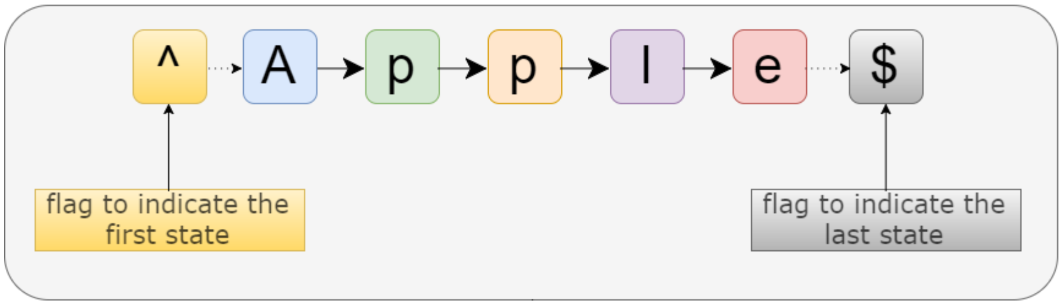
We will create a computer model of the state relationships within a body of text. This is mostly just counting on our fingers and toes. For every state (or letter) in a sequence (or word) we will look at the next state (or letter) and mark its frequency.

Consider the word ‘Apple’

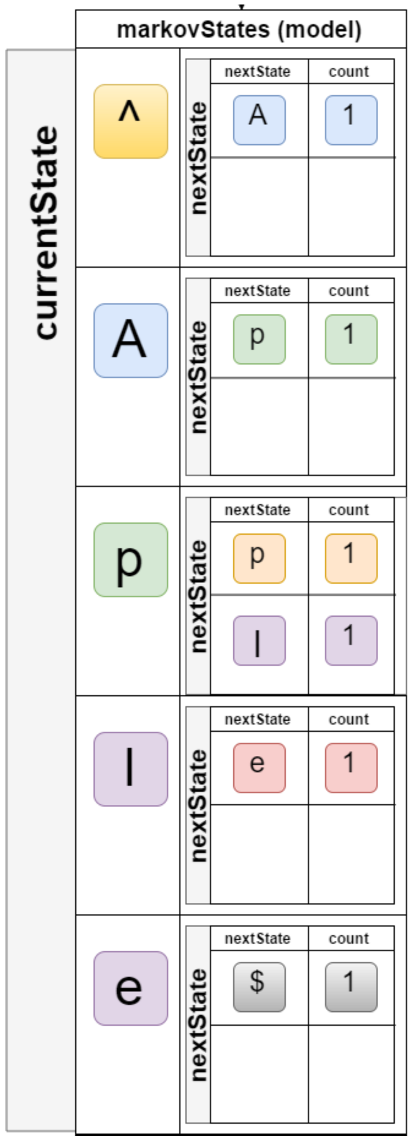
To model the letter states we would separate all the letters, e.g. {A,p,p,l,e}



Note that the letter ‘A’ starts the word, ‘A’ precedes ‘P’, ‘P’ precedes ‘P’ and ‘L,’ ‘L’ precedes ‘E,’ and ‘E’ ends the line. So we can add a flag to the beginning and end of the token, the ‘^’ and ‘$’ characters respectively, to keep track of which letters started the word and which ones ended it.

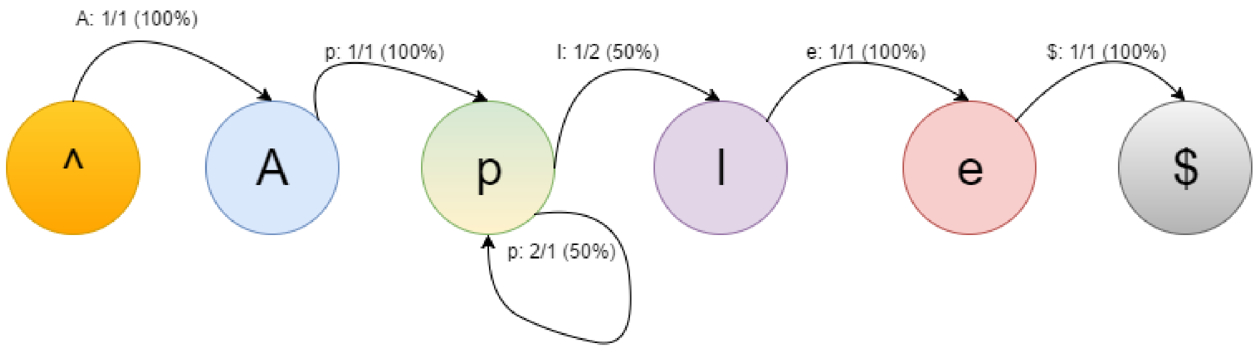

^
A
p
p
l
e
$
flag to indicate the first state
flag to indicate the last state

We will look at each character in the sequence and record which letters before and after each other. The model we are looking for will look something like this:



We count the times one state follows another. In this way, when we are at state ‘p’ we can look at the table and we see that ‘l’ follows ‘p’ 1/2 (numL / totalPossibilities) or 50% of the time. However, ‘p’ is equally likely to be followed by ‘p’ again (numP/totalPossibilities). To choose the next state, we can flip a coin, if it comes up Heads, we move on to state ‘l,’ if it comes up Tails, we return to state ‘p.’

Below is a state diagram for our mini-model.



Class & Function overview:

Here are all the required classes and functions. This is just to give you an overview of what we will be creating in the end. Do not try to create all of these in one go, it is better to follow the phases and break the requirements up.

Imports:

import java.util.HashMap;
import java.io.*;
import java.io.IOException;
import java.util.Scanner;

Main:

- Variables
    - private static FirstOrderMarkovMachine dwarfNames;
        - MarkovMachine to generate first order sequences of the “dwarf_names.txt” file.
    - private static SecondOrderMarkovMachine orcNames;
        - MarkovMachine to generate second order sequences of the “orc_names.txt” file.
    - private static SentenceMarkovMachine twoCities;
        - MarkovMachine to generate first order sequences, where the words of a sentence are the state instead of the letters themselves, of words in the ‘a_tale_of_two_cities_by_Charles_Dickens.txt’ file.
- Methods
    - public static void main(String[] args)
        - Will create three MarkovMachines (see above) and generate 5 sequences from each Machine.

abstract class MarkovMachine:

- Variables
  - public static final String FIRST_STATE_KEY = "^";
    - This is the character we will use to indicate the start of a sequence.
  - public static final String LAST_STATE_KEY = "$";
    - This is the character we will use to indicate the end of a sequence.
  - public static final int MAX_SAMPLES = 50000;
    - MAX_SAMPLES limits the number of samples we extract from a particular text file.
  - private String currentState;
    - This is the current sate at any given time of the machine. It can be thought of the ‘position’ we are at.
  - HashMap<String, HashMap<String, Integer>> markovStates;
    - This is a HashMap, which is just a Key/Value pair. The Key of this HashMap is a String which corresponds to the “current states” within the machine (‘A’, ‘p’, ‘l’, ‘e’, etc, from the Apple example above.). The Value of this HashMap is…another HashMap. The Key of this sub-HashMap will be a String that corresponds to the “next states” for this super-Key. The Value of this sub-HashMap is a frequency count. Simply, the number of times this “next state” is encountered after a particular “current state.”
- Abstract Methods
  - abstract void analyzeText(String[] tokens);
    - Text will be analyzed in different ways depending on the specific order and operation of a MarkovMachine. Takes tokens and adds their details to the markovStates model.
- Public Methods
  - Default Constructor
    - Used to initialize the local class variables.
  - public String generateSequence()
    - This method will generate and return a sequence of characters, starting with the first state (FIRST_STATE_KEY) and following the probability chain until it encounters the last state (LAST_STATE_KEY)
  - public String getNextState()
    - This method will look at the current state of the machine and randomly choose and return a ‘next state.’
  - public String getCurrentState()
    - Accessor for currentState.
  - public boolean resetState()
    - Sets currentState to correspond to the first state.
  - public HashMap<String, HashMap<String, Integer>> getMarkovStates()
    - Accessor for markovStates. It returns a deep copy of the probability model within this MarkovMachine.
- Protected Methods
  - void addStateEntry(String currentState, String nextState)

        - This method adds a ‘current state’ and a ‘next state.’ It adds these values to the markovStates HashMap.
    - String[] getTokens(Scanner fileReader)
        - This method reads the tokens (I.E. words or sentences) and returns them.
    - String[] tokenToStates(String token,int order)
        - This method takes a particular token (I.E. word or sentence) and splits it into ordered states. FirstOrderMarkovMachine, for example, will split a word (the token) into an array of single (the order) letters (the states).
- Private Methods
    - private int getSumOfValues(HashMap<String,Integer> frequencyTable)
        - Helper method that examines the sub-HashMap of a particular current state. It will look at the total count and return the sum. (i.e. in our Apple example, this function would return 1 for all letters except ‘p,’ for ‘p’ it would return a 2 because we have encountered a letter after ‘p’ two times.)
    - private int randomInt(int min, int max)
        - Helper method that returns a random integer. The method should return an integer *n* such that *min* <= *n* < *max*;

FirstOrderMarkovMachine

- Methods
    - public FirstOrderMarkovMachine(Scanner nameScanner)
        - Constructor that takes a Scanner with the text we are analyzing.
    - void analyzeText(String[] tokens)
        - Implementation of analyzeText that will create a first order probability model of inputted words.

SecondOrderMarkovMachine

- Methods
    - public SecondOrderMarkovMachine(Scanner nameScanner)
        - Constructor that takes a Scanner with the text we are analyzing.
    - void analyzeText(String[] tokens)
        - Implementation of analyzeText that will create a second order probability model of inputted words.

SentenceMarkovMachine

- Methods
    - public SentenceMarkovMachine(Scanner nameScanner)
        - Constructor that takes a Scanner with the text we are analyzing.
    - void analyzeText(String[] tokens)
        - Implementation of analyzeText that will create a first order probability distribution of inputted sentences. Unlike the other MarkovMachines, this machine will use whole words as states.
- Overloaded/Overridden Methods
    - public String generateSequence()

        - Overridden method to generate a sequence that is formatted like an English sentence.
    - String[] tokenToStates(String token)
        - Overloaded method to generate states that correspond to individual words and punctuation rather than letter series.
    - String[] getTokens(Scanner fileReader)
        - Overridden method to generate tokens that correspond to whole sentences instead of individual words.
- Private Helper Methods
    - private String textFormatter(String text)
        - Helper function that is used by tokenToStates to format/clean a token before it is separated into states.

Phase 0: markovStates HashMap

The markovStates HashMap is the center of each MarkovMachine. All the methods are either working toward creating this model, or are using the model to create sequences.

It is a HashMap (a key/value pair) with another HashMap nested as the first HashMap's value. It is declared like this: new `HashMap<String, HashMap<String,Integer>>();`

Start with this test main to see how the data structure will work. Play around with changing values, increasing and decreasing counts, etc.

```
public static void main(String[] args)
{
   // Create our nested HashMap
   HashMap<String,HashMap<String,Integer>> testMarkovStates;
   testMarkovStates = new HashMap<String,HashMap<String,Integer>>();
   System.out.println(testMarkovStates);

   // Add to our HashMap, creating the sub-HashMap while we're at it.
   testMarkovStates.put("Key1", new HashMap<String, Integer>());
   System.out.println(testMarkovStates);

   // Add to our sub-HashMap
   testMarkovStates.get("Key1").put("subKey1", 5);
   System.out.println(testMarkovStates);

   // Add to our sub-HashMap
   testMarkovStates.get("Key1").put("subKey2", 7);
```

```
        System.out.println(testMarkovStates);

        // Increment our sub-HashMap count (by overwriting previous value)
        int currentValue = testMarkovStates.get("Key1").get("subKey1");
        testMarkovStates.get("Key1").put("subKey1", currentValue + 1);
        System.out.println(testMarkovStates);
    }
```

Phase 1: Input (Main, MarkovMachine, FirstOrderMarkovMachine)

We will need three classes for phase 1. The main class, the abstract MarkovMachine class, and a child class of MarkovMachine called FirstOrderMarkovMachine. "First Order" just means that a "state" will be a single letter as in our "Apple" example above.

Main

Main will open the text file containing our name samples, use Scanner to parse them, and send the Scanner object to the constructor of FirstOrderMarkovMachine:

```
import java.util.HashMap;
import java.io.*;
import java.io.IOException;
import java.util.Scanner;


public class Markov
{

   private static FirstOrderMarkovMachine dwarfNames;
   private static SecondOrderMarkovMachine orcNames;
   private static SentenceMarkovMachine twoCities;

   public static void main(String[] args)
   {
      // Load the file of comma delimited dwarf names into a Scanner.
      File dwarfFile = new File("src/dwarf_names.txt");
      Scanner dwarfReader= null;
      try
      {
         dwarfReader = new Scanner(dwarfFile);
      }
      catch (FileNotFoundException e)
      {
         System.err.println(e.getMessage());
      }

      // Pass the scanner to our MarkovMachine for analysis.
      dwarfNames = new FirstOrderMarkovMachine(dwarfReader);
```

MarkovMachine

We will take this Object slowly.

Step1: start by getting the MarkovMachine's skeleton written:

```
abstract class MarkovMachine {
   // These variables will help us keep track of our states.
   public static final String FIRST_STATE_KEY = "^";
   public static final String LAST_STATE_KEY = "$";

   // This puts an upper limit on the number of tokens we will use.
   public static final int MAX_SAMPLES = 50000;

   // The Markov Machine only knows it's current state and possible next states.
   // This variable tracks the currentState
   private String currentState;

   // This is the Markov Probability Distribution.
   // It tracks each state and possible subsequent states
   HashMap<String,HashMap<String, Integer>> markovStates;

   // The text will be analyzed differently in each machine.
   // Primarily due to the order of the machine and delimiters.
   abstract void analyzeText(String[] tokens);

   // A default constructor for child classes to use in initialization.
   public MarkovMachine()
   {
      //Initialize Variables.
   }
```

Step 2: create a skeleton for FirstOrderMarkovMachine:

```
class FirstOrderMarkovMachine extends MarkovMachine {
   // Only Constructor for this Markov Machine. Takes in a Scanner to crunch.
   public FirstOrderMarkovMachine(Scanner nameScanner)
   {
      super();
      // Initialize markovStates, run getTokens, run analyzeText(tokens)
   }


   // This function takes in individual tokens (usually words) and creates a Markov State Table.
   void analyzeText(String[] tokens)
   {
      // This will fill the markovStates model.
   }
```

Step 3: MarkovMachine:

- String[] getTokens(Scanner fileReader)
  - Use this function to parse individual words from the Scanner object and return an array of all the words in a file.
- String[] tokenToStates(String token,int order)
  - Write a function that will take a token generated by getTokens(), separate it into individual states, and return those states as an array. The 'order' should correspond to the number of letters that make up each state. So if *token* == "Apple" and *order* == 1, tokenToStates() should return this: {"A","p","p","l","e"}. If *order* == 2 then tokenToStates() should return this: {"Ap","pl","e"}

Phase 2: Add to the Model

Step 3: MarkovMacine

- void addStateEntry(String currentState, String nextState)
  - This function adds a currentState/nextState pair to the markovStates HashMap. Note: This function will need to detect if a sub-HashMap needs to be initialized before .put() operations. In our "Apple" example from earlier, addStateEntry("A","p") would create the entry shown in the model diagram. Also, System.out.println(markovStates) would produce this output: `{A={p=1}}`
- public HashMap<String,HashMap<String, Integer>> getMarkovStates()
  - This Accessor function returns a deep copy of the markovStates HashMap.
- public String getCurrentState()
  - Accessor for currentState
- public boolean resetState()
  - Signals the MarkovMachine to start over. Hint: (change currentState's value)

Step 2: FirstOrderMarkovMachine

- void analyzeText(String[] tokens)
    - This function will fill the markovStates probability model. It uses the tokenToStates() and addStateEntry() functions to do so.
    - This function is where most of the testing will occur. To confirm that everything is functioning correctly, the markovStates Object for the dwarf_names.txt file should look like this (though the order might be a little different):

```
{A={n=1},
B={a=1, e=1, u=1, i=1, l=1, o=3},
D={a=1, u=1, w=1, o=1},
F={a=1, r=3, u=1, i=1, l=1},
G={r=2, i=2, l=1},
H={a=1},
K={i=1},
L={o=2},
N={a=3, o=2},
O={r=1, i=1, n=1},
P={o=1},
R={e=1},
T={h=3},
V={i=1},
^={A=1, B=8, D=4, F=7, G=5, H=1, K=1, L=2, N=5, O=3, P=1,
R=1, T=3, V=1},
a={r=9, i=4, l=3, n=1},
b={u=1},
d={a=1, i=1},
e={r=1, g=1, l=1},
f={a=1, u=2},
g={i=1},
h={r=2, o=1},
i={$=12, d=1, f=1, l=2, m=1, n=18},
l={a=1, i=7, o=2},
m={b=1, l=1},
n={a=4, $=17, d=1, i=1},
o={r=8, f=2, i=4, m=1, n=1},
r={a=2, $=14, e=1, i=10, o=4},
u={r=5, n=1},
w={a=1}}
```

Phase 3: SecondOrderMarkovMachine and SentenceMarkovMachine

Step 1: SecondOrderMarkovMachine

- Constructor
    - The constructor for SecondOrderMarkovMachine should be identical to FirstOrderMarkovMachine's
- void analyzeText(String[] tokens)
    - analyzeText for SecondOrderMarkovMachine is only slightly different than FirstOrderMarkov Machine. It will build a probability distribution with two letter states instead of three.

Step 2: SentenceMarkovMachine

- Constructor
    - The constructor for SentenceMarkovMachine should be identical to FirstOrderMarkovMachine’s
- void analyzeText(String[] tokens)
    - analyzeText will still be only concerned about states. It should be very similar to analyzeText from FirstOrderMarkovMachine and SecondOrderMarkovMachine
- String[] getTokens(Scanner fileReader)
    - getTokens will return an array of all *sentences* in the text file, rather than all individual words. (Hint: fileReader.useDelimiter)
- String[] tokenToStates(String token)
    - tokenToStates will return an array of all the words in an individual sentence. (Note: You will need to format the text before splitting it up)
- private String textFormatter(String text)
    - This method will clear out new-line characters and handle “in sentence” punctuation. E.g. commas. Format this special punctuation in a way that allows tokenToStates to interpret it as a separate state.

Phase 4: Running the machine.

Step 1: MarkovMachine()

- public String getNextState()
    - This method will look at the machine’s current state (*currentState*) and probabilistically choose a next state. This next state is returned and becomes the new *currentState.* Note: this function will use getSumOfValues() and randomInt methods to choose the next state. If currentState == “p” and our markovStates model were the same as in our “Apple” example above, getNextState() should return “p” about 50% of the time and “l” about 50%.
- private int getSumOfValues(HashMap<String,Integer> frequencyTable)
    - This method will examine a sub-HashMap from the markovStates table and return the sum of the counts (the Integers).
- private int randomInt(int min, int max)
    - Helper method that returns a random integer. The method should return an integer *n* such that *min* <= *n* < *max*;
- public String generateSequence()
    - This method will run the MarkovMachine trough a whole sequence, from beginning (FIRST_STATE_KEY) to end (LAST_STATE_KEY) and returns the collected sequence.

Step 2: SentenceMarkovMachine()

- public String generateSequence()

    - This method is similar to the generateSequence method in MarkovMachine, but we override it here so that the generated sequence looks like an English sentence. E.g. spaces in between words and punctuation at the end.

Step 3: Main

- The final main() will create 3 MarkovMachines and generate 5 sequences from each of them.
    - private static FirstOrderMarkovMachine dwarfNames;
        - This MarkovMachine will examine the “dwarf_names.txt” file.
    - private static SecondOrderMarkovMachine orcNames;
        - This MarkovMachine will examine the “orc_names.txt” file.
    - private static SentenceMarkovMachine twoCities;
        - This MarkovMachine will examine the “a_tale_of_two_cities_by_Charles_Dickens.txt” file.
- Here is a sample output from the final main():

```
Here are your 5 Tolkien-inspired Dwarf names:
Bloroi
Finarofur
Vin
Gin
Fr

Here are your 5 Tolkien-inspired Orc names:
Gazrat
Lungash
Muzlun
Radbag
Ughur

Here are your 5 sentences written in the style of Charles Dickens:
It is likely enough that favoured period, that sufferer was the superlative degree of
loaves and seventy-five.

Mere messages, it was the chickens of whom a dirty procession of the sublime
appearance by pigs, it was the age of the epoch of monks which passed within his view,
it was the period, rude carts, and a congress of France.

Mere messages, rolled with a king with a king with a queen with such humane
achievements as to the season of the chickens of incredulity, it, terrible in the age
of times, it was the superlative degree of monks which, to death, had everything
before us, we had already marked by announcing that arrangements were made for ever.

There were settled for evil, that arrangements were growing trees, it was the epoch of
some fifty or for good or sixty yards.

It is likely enough that sufferer was the period was the Life Guards had recently
attained her five-and-twentieth blessed birthday, it was the Life Guards had recently
attained her five-and-twentieth blessed birthday, it was the lords of this.
```