Assignment 4: Markov Model of Natural Language

The University of British Columbia
Science One CS 2015-2016
Instructor: Mike Gelbart

Due Friday, March 18 at 5:00pm

Last updated March 1, 2016

Perspective. In the 1948 landmark paper A Mathematical Theory of Communication, Claude Shannon founded the field of information theory and revolutionized the telecommunications industry, laying the groundwork for today's Information Age. In this paper, Shannon proposed using a Markov chain to create a statistical model of the sequences of letters in a piece of English text. Markov chains are now widely used in speech recognition, handwriting recognition, information retrieval, data compression, and spam filtering. They also have many scientific computing applications including the genemark algorithm for gene prediction, the Metropolis algorithm for measuring thermodynamical properties, and Google's PageRank algorithm for Web search. For this assignment, we consider a whimsical variant: generating stylized pseudo-random text.

Markov model of natural language. Shannon approximated the statistical structure of a piece of text using a simple mathematical model known as a Markov model. A Markov model of order 0 predicts that each letter in the alphabet occurs with a fixed probability. We can fit a Markov model of order 0 to a specific piece of text by counting the number of occurrences of each letter in that text, and using these frequencies as probabilities. For example, if the input text is "gagggagaggcgagaaa", the Markov model of order 0 predicts that each letter is "a" with probability 7/17, "c" with probability 1/17, and "g" with probability 9/17 because these are the fraction of times each letter occurs. The following sequence of characters is a typical example generated from this model:

g a g g c g a g a a g a g a a g a a a g a g a g a g a a a g a g a a g...

A Markov model of order 0 assumes that each letter is chosen independently. This independence does not coincide with statistical properties of English text because there is a high correlation among successive characters in a word or sentence. For example, "w" is more likely to be followed with "e" than with "u", while "q" is more likely to be followed with "u" than with "e". We obtain a more refined model by allowing the probability of choosing each successive letter to depend on the preceding letter or letters. A Markov model of order k predicts that each letter occurs with a fixed probability, but that probability can depend on the previous k consecutive characters. Let a k-gram mean any string of k characters. Then for example, if the text has 100 occurrences of "th", with 60 occurrences of "the", 25 occurrences of "thi", 10 occurrences of "tha", and 5 occurrences of "tho", the Markov model of order 2 predicts that the next letter following the 2-gram "th" is "e" with probability 3/5, "i" with probability 1/4, "a" with probability 1/10, and "o" with probability 1/20.

A brute-force solution. Claude Shannon proposed a brute-force scheme to generate text according to a Markov model of order 1:

"To construct [a Markov model of order 1], for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage."

Your task is to write a Python program to automate this laborious task in a more efficient way—Shannon's brute-force approach is prohibitively slow when the size of the input text is large.

Structure of your program. Write a Python program `markov.py` that takes in three command-line arguments: the order k (an integer), the size of the output, T (an integer), and the name of the file containing the training text (a string). Please read the command line arguments in this order (k, then output size, then file name). Your program should perform the following operations:

- **Read the text from the file**. If the name if the file is stored in the variable `filename`, you can read its contents into the string `text` using the following code:

  ```
  with open(filename) as f:
      text = f.read()
  ```

- **Count the occurences of k-grams**. Loop through the training text and, at each position, record the k-gram as well as the following letter. The results will be stored in a dictionary whose keys will be the k-grams, and whose values will be lists of all characters following the k-gram. This operation should be performed as if the text were circular (i.e., as if it repeated the first k characters at the end); you can achieve this by explicitly adding the first k characters to the end of the string, using the following code: `text += text[:k]`.

  Let's return to our example of the input text "gagggagaggcgagaaa". Let us assume $k = 2$. We then record, for each 2-gram, the following character. The following table represents what the dictionary should look like:

k-gram	next char
aa	a, g
ag	g, a, g, a, a
cg	a
ga	g, g, g, g, a
gc	g
gg	g, a, c

  Table 1: Dictionary generated from the input string "gagggagaggcgagaaa".

  It is highly recommended that you re-create this table by hand with pencil and paper, and make sure your table matches the one shown here. This will be a good sign that you understand the task at hand. Note that the table above accounts for the circularity of the text; this is why, for example, it indicates that the character "g" is considered to follow the 2-gram "aa".

- **Generate pseudo-random text.** Your output text should start with the same k characters as the training text starts with. Then, add $T - k$ additional characters to your new text, one at a time. To do this, randomly select each new character based on the preceding k-gram, by looking up this k-gram in your dictionary and then picking randomly from the list of possibilities. You can pick a random element from a list using `numpy.random.choice`. As in Assignment 1 (Checkerboard), you can use `sys.stdout.write` to print to the screen without ending the line.

Style guidelines. Your program will graded on both correctness and style. Regarding style, try to use reasonable variable names and add comments where necessary. You do not need to organize your program into functions, but if you do, make sure to document the inputs and outputs of each function.

Experimentation. Once you get the program working, test it on different inputs of different sizes and different values of k. Does increasing k have the effect you expect? Try your model on something that you have written or some other text you know well. Make sure to test both long inputs (we provide several) and long outputs.

Example outputs. Some example outputs are provided for your reference, assuming the file `text17.txt` contains the string "gagggagaggcgagaaa".

```
>> python markov.py 2 11 input17.txt
gaggcgagaag

>> python markov.py 2 11 input17.txt
gaaaaaaagag
```

Here is a longer example, trained on a combination of a speech by Justin Trudeau (World Economic Forum, January 2016) and an excerpt from *As You Like It* by William Shakespeare, using $k = 7$:

> ACT II
>
> [All together here]
>
> And she believe leadership on climate change equal in magnitude and universal theatre Presents more woeful ballad Made to his mistress hath ta'en a hurt, Yea, providently caters for those diversity as a source of strength, not a coincidence that I were a fool Much marked of the fourth industrial revolution.
>
> What a breathtaking possible to relieve any thing. I would have hired Dominic Barton at one point or another.
>
> We have a diverse ways of seeing and thinking come to me.
>
> CELIA I pray you, bear with me, That little recks to find him like a doe, I go to find the cow's dugs that then? Let me see wherein My tongues.
>
> ORLANDO Nor shalt have libertine, And in my heart to disgrace my man's apparel and to your warm welcome and for bring again toward a low-carbon economy will produce mass unemployment and greater inequality.
>
> Technologies could produces tangible results we achieve for people, the more fool I; when I welcomed them

Deliverables. Submit the following files on Connect:

1. Your code, in a file called `markov.py`
2. A file called `output.txt` containing one of the most entertaining language-modeling fragments that you discover (as well as a description of the input file used and the value of k). If you wish, you can combine the training text files to produce more interesting results.
3. A file called `readme.txt` containing the answers to the following questions, using big-O notation:
   (a) What is the asymptotic running time of your program, as a function of N and T, where N is the length of the training text and T is the size of the output text? Assume that reading/writing to the dictionary and calling `numpy.random.choice` both take $O(1)$ time.
   (b) What is the asymptotic memory usage of your program, as a function of N and T, where N is the length of the training text and T is the size of the output text?
4. **(Optional)** Your extra credit file `markov2.py` (see below).

Collaboration. Please make sure you are familiar with the Collaboration Policy on the course website.

Extra credit. The implementation described above is very inefficient in its memory usage. Write a new program `markov2.py` that uses nested dictionaries to count the *number of occurrences* of each letter following each k-gram, instead of (redundantly) storing them all in a list. In other words, the values of your main dictionary should themselves be dictionaries, whose keys are the possible next characters for that k-gram, and whose values are integers representing the number of times this character follows the k-gram. Then, to generate text, compute the probabilities of each possible next letter given the k-gram and pass in the probabilities to `numpy.random.choice` using the syntax `np.random.choice(characters, p=probs)`, where `characters` is a list containing the characters and `probs` is a list (or NumPy array) of the same length, containing the corresponding probabilities. The idea here is that if you modified your program to read the file piece-by-piece (no need to do this!), it could operate on training text files so large that they could not even fit into your computer's memory (say, for example, all of Wikipedia).

Attribution. Assignment adapted with permission from Princeton COS 126, "Markov Model of Natural Language" at `http://www.cs.princeton.edu/courses/archive/fall15/cos126/assignments/markov.html`. Original assignment was developed by Bob Sedgewick and Kevin Wayne.