University of Colorado Boulder
APPM 3310 - Matrix Methods and Applications
Fall 2016

Markov Chains with Applications to Text Generation

Josh Jacobson
104260304
Section 002

Michael Voecks
102584009
Section 002

May 3, 2017

Abstract

As branch of probability theory and statistics, the study of Markov chains is fundamentally based on practices in linear algebra. This paper applies those practices in the basic description of Markov chains, especially concerning how they are applied to random text generation. After outlining the mathematical framework, we demonstrate the development of a text generation algorithm using first, second, third, and fourth-order Markov chains. With our resulting examples, we successfully demonstrate that higher-order Markov chains are able to more accurately generate random output text, similar in syntax and style to the given input text.

Contents

1 Introduction

Named after the Russian mathematician Andrey Markov, a Markov chain is a stochastic process that has direct relations to probability theory and statistics. Markov chains have many applications as statistical models of real-world processes, making them a very interesting and highly valuable topic of study. Some examples include, game theory, speech recognition, and mathematical biology. In this paper, we are specifically interested in the probability theory aspect and the application of Markov chains to text generation. In addition, this paper outlines the development of an algorithm for processing input text using Markov chain analysis to output new random text that exhibits similar syntax and style. We then offer an example solution in the form of Python script, using passages from Lewis Carroll's *Alice in Wonderland* as experimental input text.

This paper operates on a general knowledge of concepts in linear algebra and data structures, such as the use of eigenvectors, matrix multiplication, and computational dictionaries. As it would be beneficial for readers to be familiar with basic probability and set theory, we provide a brief background on conditional probability and the discrete inverse transform for simulating discrete random variables. Otherwise, this paper lays the mathematical foundation for the use of Markov chains with probability transition matrices and integrates these methods into applications for text generation. We ultimately compare algorithms using Markov chains of increasing order to test whether greater reliance on previous states improves the accuracy of the random output text.

2 Markov Chains and Linear Algebra

2.1 Mathematical Symbols

Below is a list of the nomenclature used in this report by order of appearance:

Abbreviation	**Description**
X_t	the state of the chain X at time t
p	the probability transition matrix
$p(i, j)$	the entry of p in row i, column j
$P(A = t)$	the probability of event A having the value t

2.2 Simplifying Assumptions

For the duration of this paper, unless specifically noted, a Markov chain will refer to a *homogeneous, first order Markov chain*. A homogeneous Markov chain is time-homogeneous and does not rely on the specific time of a series of states to refer to the prediction of future states. A first order Markov chain is a chain that relies solely on the current state of the chain when making predictions about future states of the chain.

2.3 Background

In this paper, we refer to transitions, or probability transitions frequently. A *probability transition* from one state to another refers to the conditional probability of ending at one state having started at another state. *Conditional probability* is a measure of the probability of an event B occurring, with the knowledge that a separate event A has already happened. Conditional probability is represented mathematically as $P(B|A)$, where we are measuring the probability of event B knowing all information pertaining to event A.

In the application of Markov chains to text generation, this paper develops an algorithm using simulated Markov chains to generate random text. Because computers are only able to effectively simulate a random number between 0 and 1 we use the discrete inverse transform to simulate events throughout the algorithm. The discrete inverse transform is a method of accurately simulating a discrete random variable using only a random number between 0 and 1. The *discrete inverse transform* method is defined as follows

Let $F(x)$, $x \in \mathbb{R}$, denote any cumulative distribution function (cdf) (continuous or not). Let the inverse function be defined by

$$F^{-1}(y) = \min\{x : F(x) \geq y\}, y \in [0, 1]$$

Define $T = F^{-1}(U)$, where U has the continuous uniform distribution over the interval (0,1). Then T is distributed as F, that is, $P(T \leq x) = F(x)$, $x \in \mathbb{R}$ [4].

2.4 Markov Chain Basics

In many ways, the study of Markov chains relies on practices in linear algebra. The process involves making predictions of future events based off previous *chain states*. The list of all possible states is then the system's "state space". Traditionally, a Markov chain is represented as a square matrix whose columns and rows are spanned by its state space, where each (i, j) entry denotes the probability that state $n + 1$ will be j given that state n was i. Then we have the square matrix

$$p(i, j) = P(X_{n+1} = j|X_n = i). \tag{1}$$

This matrix is called the *probability transition matrix* of the Markov chain [3]. An example of a Markov chain is given below.

2.4.1 An Illustrative Example

Consider the three friends, Mark, John, and Stacy playing catch. Mark only throws the ball to Stacy, Stacy will throw the ball to John or Mark with equal probability, and John will throw the ball to Mark with probability 3/4, or to Stacy with probability 1/4. A person must throw the ball to somebody else, and cannot pass to themselves. We want to construct the Markov chain that represents who has the ball at any given time [2].

This situation is represented in the directed graph below, where each node of the graph is a possible state of the chain, the arrows between nodes represent a transition from one state to another, and the arrow weighting represents the probability that a transition from one state to another will occur. Graphical representations of Markov chains are often useful as visual aids. For example,

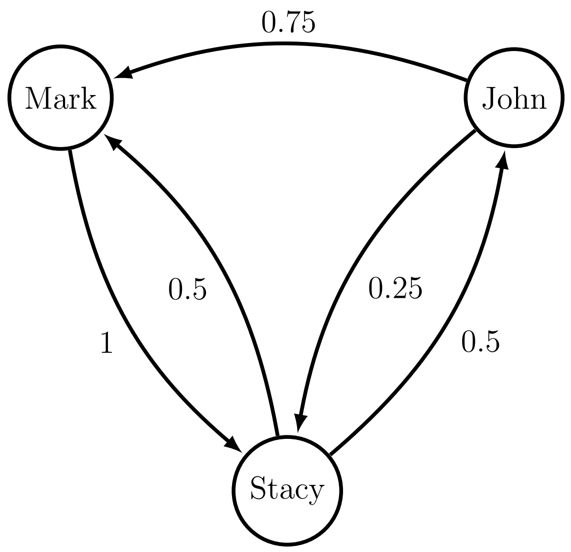

Figure 1: A graphical representation of a Markov chain

Recall that the state space of a matrix is all the possible states that the system can hold. In this case, the state space refers to the person holding the ball. Because there are three states in this state space, the probability transition matrix that represents the Markov chain is given by the 3×3 matrix

$$\mathbf{p} = \begin{array}{c} \\ M \\ J \\ S \end{array} \begin{array}{c} \begin{array}{ccc} M & J & S \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0.75 & 0 & 0.25 \\ 0.5 & 0.5 & 0 \end{bmatrix} \end{array} \tag{2}$$

Again, recall that the entries of the probability transition matrix denote the probability of transitioning to one state given the current state. In row M, denoting Mark, we see that because he only passes to Stacy, the entry (M, S) is 1. However, in row S, we see that Stacy passes to Mark with probability 0.5 and to John with probability 0.5, so she passes to them both with equal probability.

Notice that all rows of the probability transition matrix sum to 1. In probability theory, the probability of all possible events must sum to 1 as some event must happen even if that event denotes nothing changing. So because each row represents all possible transitions of an event, all rows in a probability transition matrix must sum to 1.

2.5 N-Step Probability Transitions

Now that we have established the basic principles of Markov chains and how to accurately represent them, we continue with a discussion of their relation to linear algebra. First, recall the notion of the probability transition matrix, or the probability that event j will occur given that event i has just occurred. This is represented by the (i, j) entry in the probability transition matrix p. Now we extend this notion to consider the probability of event j occurring in two, three, or n transitions, given that event i has just happened. To compute the n-step probability transition from i to j, we take the probability transition matrix and raise it to the n^{th} power. So the entry $p^n(i, j)$ corresponds to the n^{th}-step probability transition from i to j. This process is proven using induction [3].

We call p the *first-step* probability matrix. In general, the n^{th}*-step* probability matrix is the denoted as $p^{(n)}$ with entries

$$p^{(n)}(i,j) = P(X_n = j|X_0 = i) \tag{3}$$

Theorem. *$p^{(n)} = p^n$, i.e. the n^{th} power of p.*

Proof. By induction on n, the number of transitions.
Basis: $n = 1$: $p^{(1)}(i,j) = P(X_1 = j|X_0 = i) = p(i,j)\ \forall i,j$. So, $p^{(1)} = p$.
Induction hypothesis: Assume $p^{(n)} = p^n$. We show $p^{(n+1)} = p^{n+1}$
Induction:

$$\begin{aligned}
p^{(n+1)}(i,j) &= P(X_{n+1} = j|X_0 = i)\\
&= \sum_{z\in S} P(X_{n+1} = j, X_n = z|X_0 = i)\\
&= \sum_{z\in S} P(X_{n+1} = j|X_n = z, X_0 = i)\cdot P(X_n = z|X_0 = i)\\
&= \sum_{z\in S} P(X_{n+1} = j|X_n = z)\cdot p^{(n)}(i,z)\\
&= \sum_{z\in S} p(z,j)\cdot p^n(i,z) \quad *\\
&= (p^n\cdot p)(i,j)\\
&= p^{n+1}(i,z)
\end{aligned}$$

where $z \in S$ denotes that z is an element in the state space. The $(*)$ notes the use of the inductive hypothesis. □

2.6 Limit Behavior

We now consider what happens when n in an n^{th}-order probability transition matrix becomes very large. First, however, we must address the *stationary distribution* of a Markov chain.

A stationary distribution is a row vector that satisfies the equation $\pi p = \pi$, and with entries that sum to 1. The stationary distribution can be thought of as the left-eigenvector of the probability transition matrix. If a Markov chain has the distribution π at time $t = n$, then it will retain that distribution for all time $t \geq n$. For this reason, the stationary distribution is often called the "steady state" of the Markov chain. When a Markov chain has a unique stationary distribution π, then

$$\lim_{n\to\infty} p^n(i,j) = \pi(j) \tag{4}$$

This implies that for a sufficiently large n, the n^{th}-step probability transition from i to j will be independent of i, and will only rely on the j^{th} entry of the stationary distribution. Furthermore, this implies that the fraction of time a chain spends at a given state i as time tends towards infinity is equal to the i^{th} entry of the stationary distribution [3].

2.7 Higher Order Markov Chains

Thus far, this paper has solely been concerned with first-order Markov chains, or chains that only rely on the present state in determining the next state. Higher order Markov chains make predictions about future states based not only on the present state of the chain, but past states as well. For higher order Markov chains, we consider the probability transitions to state a_n, where we know the previous states a_{n-1}, a_{n-2}, ... with

$$P(X_n = a_n | X_{n-1} = a_{n-1}, X_{n-2} = a_{n-2}, \ldots). \tag{5}$$

Because we now require more information about the history of the chain to determine the next transition, it becomes more difficult to represent higher order Markov chains in the traditional matrix format. However, higher order Markov chains can be represented as first-order Markov chains where the cluster of all relevant previous states is treated as a single state, or the "condensed" state. Note that in an n^{th}-order Markov chain,

$$P(X_i | X_{i-1}, X_{i-2}, \ldots X_1) = P(X_i | X_{i-1}, \ldots X_{i-n}). \tag{6}$$

For example, an n^{th}-order Markov chain over some set A is equivalent to a first-order Markov chain over the set A^n of $n-tuples$. Higher order Markov chains operate with more "history" and thus provide better predictive value [1].

3 Model Implementation and Results

3.1 Text Generation with Markov Chains

Clearly, Markov chains can be a very powerful and very useful tool, especially for making predictions. To illustrate a powerful application of Markov chains, we have developed and implemented an algorithm for randomly generating text in the same "syntax" and "style" as a given input text by processing the text through a Markov chain of word associations. The algorithm reads a given input text file to create a list of individual words. From this list, we generate a probability transition matrix, the state space of which is spanned by the words of the input corpus. The entries of the probability transition matrix are then the probability that a given word in the state space will follow the previous word. For the purpose of memory allocation and simplicity, the probability transition matrix is implemented using a dictionary–a data structure that stores information in key-value pairs. The state space of the probability transition matrix makes up the keys of the dictionary, and the values associated to each key are arrays of all the words that follow the key in the corpus. In the following sections, we outline our algorithm for text generation in detail and compare the results of implementation with first, second, third, and fourth order Markov chains.

3.1.1 A Markov Algorithm

The algorithm is split into two major sections, the first of which populates the probability transition matrix, in this case a dictionary, and the second which generates the random text from the probability transition matrix.

Part I:

1. Create a plaintext file containing the input text to be analyzed by the algorithm.
2. Variable Initiation: Establish the following variables
   - **lastWord**—set *lastWord* equal to the first word in the plaintext file; this variable will be continuously updated to hold the last word processed
   - **ptm**—the dictionary corresponding to the probability transition matrix
3. Iterate through each word in the corpus, convert it to lowercase, and place it in the location of the key *lastWord* in the *ptm* dictionary as it is processed.
4. Update the value of *lastWord* to the current word being processed, this way, when the program loops through for the next word, the previous word will be correct.
5. Repeat steps 3–4 until all words in the file are processed.

Part II:

1. Select the first word to be printed by choosing a random value from the *ptm* dictionary and print the word.
2. Using the last word that was printed as the lookup key, choose a random value from the *ptm* dictionary to print out the next word. This is achieved by computing the equivalence of a discrete inverse transform on all possible choices for the lookup key and selecting one at random.
3. Repeat step 2 until the desired amount of randomly chosen words have been printed.

This is a Markov algorithm in that the last word is the current state, the next word depends on the last word only—or on the *present state* only—and the next word is randomly chosen from a statistical model of the corpus. The likelihood of each random word choice can then be represented as the probability transition matrix.

The above algorithm outlines a method for generating random text using a first-order Markov chain model. To illustrate how the use of higher order Markov chains in text generation leads to more representational outputs, we have also implemented the algorithm using second, third, and fourth order Markov chains. The main difference between the algorithms is how the *lastWord* variable is created. For second order, it contains the previous 2 words, for third order it contains the previous 3 words, and for fourth order it contains the previous 4 words (see appendix). Note that the algorithm operates on the entire input corpus, but generates an output of a user specified length.

3.1.2 Variable-Order Chain Comparisons

For the sake of demonstration and comparison, we have tested the algorithm on the entirety of Lewis Carroll's *Alice in Wonderland*. Below, we provide an original sample of the text for reference on style, followed by 100 word outputs of the algorithm using Markov chains of increasing order.

> **Sample:**
> Alice was just beginning to think to herself, 'Now, what am I to do with this creature when I get it home?' when it grunted again, so violently, that she looked down into its face in some alarm. This time there could be no mistake about it: it was neither more nor less than a pig, and she felt that it would be quite absurd for her to carry it further. So she set the little creature down, and felt quite relieved to see it trot away quietly into the wood.
>
> **First-Order:**
> of the wandering hair that will be clearer than ever,' thought she began looking round. (it was only changing so she wasn't asleep,' he doesn't matter,' it was now for anything but there were saying to go round!" ' 'somebody said,' alice got to go down—here, bill! the king, going to invent something!' 'i—i'm a caucus-race and dogs.' it chose to the little anxiously. 'yes,' said the other.' as the glass table to go through,' thought alice. the hatter, and, what's that beautiful soup! soup does the mock turtle sighed the poor alice! it off. 'give your hat,'
>
> **Second-Order:**
> for his housemaid,' she said to herself, in a coaxing tone, and added 'it isn't a bird,' alice remarked. 'right, as usual,' said the mock turtle to the general conclusion, that wherever you go on? it's by far the most curious thing i know. silence all round, if you hold it too long; and that in the shade: however, the multiplication table doesn't signify: let's try the effect: the next moment she quite forgot how to get in?' 'there might be some sense in your pocket?' he went on eagerly: 'there is such a thing before, and behind it

Third-Order:
no meaning in it.' the jury asked. 'that i can't be civil, you'd better ask her about it.' 'she's in prison,' the queen was silent. the king repeated angrily, 'or i'll have you been doing here?' 'may it please your majesty,' said the caterpillar. alice said very humbly; 'i won't indeed!' said alice, 'a great girl like you,' (she might well say this), 'to go on for some way, and nothing seems to like her, down here, that i should understand that better,' alice said with some difficulty, as it was a queer-shaped little creature, and held it out

Fourth-Order:
questions.—how am i to get in?' she repeated, aloud. 'i shall sit here,' he said, 'on and off, for days and days.' 'but what am i to get in?' 'there might be some sense in your knocking,' the footman went on without attending to her, 'if we had the door between us. for instance, if you were inside, you might knock, and i could let you out, you know.' he was looking up into the sky all the time at the thought that it might be hungry, in which case it would be quite as much as serpents do,

As expected, the output text of each algorithm is gibberish, however, representational improvement can clearly be seen as the order of the Markov chain used increases. With the first-order Markov chain, the output appears to be a completely random selection of words, but by the fourth-order Markov chain we see segments resembling actual sentences.

Note that in these somewhat lower order cases, the generated text does not appear in the same way anywhere in the book. However, as the order of the Markov chain used continues to increase, the algorithm would reach a threshold at which it would return selections pulled directly from the text. This happens when the order of the Markov chain used exceeds the magnitude of the input corpus. In this sense, lower order Markov chains are better for use with small corpora, where higher order Markov chains are better for use with large corpora.

4 Final Remarks

4.1 Strengths and Weaknesses

This algorithm was designed with the intention of being simple, yet accurate. The two stage programming process leads to a set of strengths and weaknesses of the overall model:

Strengths:

- The state can be set to depend on one word, two words, or any number of words. As the number of words in each state increases, the generated text becomes less random.
- Order of the markov chain used can be varied relative to the size of the input text to generate better looking outputs.

- The larger the input text, the more choices there are at each transition, generating a better looking output.

Weaknesses:

- If the order of the Markov chain used is too high relative to the magnitude of the input text, the output will be very similar, if not exact in resemblance to the input.
- The algorithm involves stripping the capitalization of letters such that capitalized words do not appear mid-sentence. Unfortunately, this detracts from the authenticity of the generated text.

4.2 Future Model Development

While the algorithm is already highly functional, there are a few improvements we could look into to improve the generated text.

- **Chain Order–Corpus Magnitude Ratio:** We would like to establish some method of calculating the ideal chain order to corpus magnitude ratio such that the appropriate chain is automatically chosen to generate text in recognizable syntax and style without becoming identical to the input.
- **Output Formatting:** We would like to be able to include capitalized letters in such a manner that they only appear at the beginning of a sentence, as well as distribute punctuation more naturally in general. This may involve special consideration or weightings for capitalized words in the building of the dictionary.

4.3 Conclusions

Applying what we have learned about Markov chains and their necessary relation to methods in linear algebra, we have developed an algorithm for accurately generating new random output text from a given input text. The algorithm is a two step process which first involves building a dictionary from the given input, and then processing this dictionary into a new sequence using Markov chain word associations. As there will be more choices at each transition, a larger input text will always yield better looking output text. However, because the size of the input text cannot always be controlled, the relative order of the Markov chain used in the algorithm can be adjusted to better correspond to the size of the input text.

Now that we have established a method for generating text from a given corpus, we would be interested in researching topics where text generation is useful, as well as how our algorithm could be better optimized and applied to each of these topics.

5 Appendix

5.1 Python Algorithms

The same algorithm was used for each n-order Markov chain, with the only difference in coding being the treatment of the *lastWord* variable. We have provided the entire script for a first-order Markov chain, where for the the second, third and fourth order chains we have included only the differences in the use of the *lastWord* variable.

5.1.1 First-Order Script

```
#This file generates random text based on an input text using a first
#order markov chain model

#import necessary libraries
import random
import sys

#define the probability transition matrix hashtable and the lastword variable
lastWord = ''
ptm = {}

#This section populates the probability transition matrix by scanning every
#word of the document and updates the entries of the corrisponding words in
#the hashtable so that its rows are all the words in the document and their
#columns consist of all the words that follow that specific word

#these lines get the next word from the text
with open('sample.txt', 'r') as f:
    for line in f:
        for word in line.split():

            #the word is first converted to lowercase and checked to see
            #if it is actually a word
            word = word.lower()
            if word != '':

                #this first if statement will only return true once at the
                #very start of the algorithm, and just sets the first word of
                #the document to the variable lastWord
                if lastWord == '':
                    lastWord = word
                #For every subsequent word in the document the following
                #code runs
                else:
                    #if the previous word is not in the probability transition
                    #hashtable then add it and make its only entry the word
                    #we are processing
                    if not(lastWord in ptm):
                        ptm[lastWord] = [word]
                    #if the previous word exists in the probability transition
                    #hashtable then update its columns to include the word
                    #we are currently processing
                    else:
                        ptm[lastWord].append(word)
                    #set the variable lastWord to the word currently being
                    #processed so the next loop associates the correct words
                    lastWord = word

#This section of the code generates the text by picking a random starting word,
#and then based of that random starting word it picks the next word randomly
#from the previous words probabiliy transition hashtable

#pick a random word from the hashtable
start = random.choice(list(ptm))

#loop until the user specified number of words have been randomly generated
for i in range(1, int(sys.argv[1])):
    #choose a random word to be printed next based of the previous word
    word = random.choice(ptm[start])
    #reset the previous word to be the current word that was just picked
    start = word
    #print out the word set that was set aside for printing
    sys.stdout.write(word+' ')
    sys.stdout.flush()
print()
```

5.1.2 Second-Order Script

```
#Define the last word to be an array of two words to represent the last
#two word in the text
lastWord = ('', '')
ptm = {}
with open('sample.txt', 'r') as f:
    for line in f:
        for word in line.split():
            word = word.lower()
            if word != '':
                #the first if and elif sets the lastWord variable to be
                #the first two words of the text
                if lastWord[0] == '':
                    lastWord = (word, '')
                elif lastWord[1] == '':
                    lastWord = (lastWord[0], word)
                else:
                    if not(lastWord in ptm):
                        ptm[lastWord] = [word]
                    else:
                        ptm[lastWord].append(word)
                    #The lastword variable still includes the most recent
                    #word and the currennt word, however the most recent
                    #word is shifted one position down.
                    lastWord = (lastWord[1], word)
```

5.1.3 Third-Order Script

```
#Define the lastWord varibale to be an array of the last three
#variables in the text.
lastWord = ('', '', '')
ptm = {}
with open('sample.txt', 'r') as f:
    for line in f:
        for word in line.split():
            word = word.lower()
            if word != '':
                #again populate the lastword variable with the first three
                #words in the text
                if lastWord[0] == '':
                    lastWord = (word, '', '')
                elif lastWord[1] == '':
                    lastWord = (lastWord[0], word, '')
                elif lastWord[2] == '':
                    lastWord = (lastWord[0], lastWord[1], word)
                else:
                    if not(lastWord in ptm):
                        ptm[lastWord] = [word]
                    else:
                        ptm[lastWord].append(word)
                    #again, the 2 most recent words are shifted down one
                    #position and the new word is added to the lastWord
                    #variable
                    lastWord = (lastWord[1], lastWord[2], word)
```

5.1.4 Fourth-Order Script

```
#Define the lastWord variable to contain an array of the 4 most recent words
lastWord = ('', '', '', '')
ptm = {}
with open('sample.txt', 'r') as f:
    for line in f:
        for word in line.split():
            word = word.lower()
            if word != '':
                #populate the lastWord variable to contain the first four
                #words to start
                if lastWord[0] == '':
                    lastWord = (word, '', '', '')
                elif lastWord[1] == '':
                    lastWord = (lastWord[0], word, '', '')
                elif lastWord[2] == '':
                    lastWord = (lastWord[0], lastWord[1], word, '')
                elif lastWord[3] == '':
                    lastWord = (lastWord[0], lastWord[1], lastWord[2], word)
                else:
                    if not(lastWord in ptm):
                        ptm[lastWord] = [word]
                    else:
                        ptm[lastWord].append(word)
                    #Shift the position of the 3 most recent words and add
                    #the word currently being processed
                    lastWord = (lastWord[1], lastWord[2], lastWord[3], word)
```

References

[1] Mark Craven. Markov chain models 2. `https://cw.fel.cvut.cz/wiki/_media/courses/a6m33bin/markov-chains-2.pdf`, 2011. [Online Lecture Slides; accessed December 3, 2016].

[2] Dartmouth College Teaching Aid. Chapter 11 Markov Chains. `https://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf`. [Online Book Chapter; accessed November 26, 2016].

[3] Richard Durrett. *Essentials of Stochastic Processes*. Springer, New York, NY, 2012.

[4] Karl Sigman. Inverse transform method. `http://www.columbia.edu/~ks20/4404-Sigman/4404-Notes-ITM.pdf`, 2010. [Online Notes; accessed December 6, 2016].