# 1 Markov Chains

**Lab Objective:** *A Markov chain is a collection of states with specified probabilities for transitioning from one state to another. They are characterized by the fact that the future behavior of the system depends only on its current state. In this lab we learn to construct, analyze, and interact with Markov chains, then use a Markov-based approach to simulate natural language.*

## State Space Models

Many systems can be described by a finite number of *states*. For example, a board game where players move around the board based on dice rolls can be modeled by a Markov chain. Each space represents a state, and a player is said to be in a state if their piece is currently on the corresponding space. In this case, the probability of moving from one space to another only depends on the player's current location; where the player was on a previous turn does not affect their current turn.

Markov chains with a finite number of states have an associated *transition matrix* that stores the information about the possible transitions between the states in the chain. The $(i, j)$th entry of the matrix gives the probability of moving **from state** $j$ **to state** $i$. Thus, each of the columns of the transition matrix sum to 1.

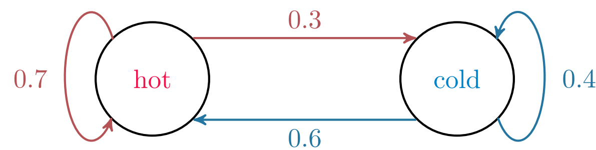
> NOTE
>
> A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the $(i, j)$th entry of the matrix is the probability of moving from state $i$ to state $j$. Both representations are common, but in this lab we exclusively use column stochastic transition matrices for consistency.

Consider a very simple weather model in which the weather tomorrow depends only on the weather today. For now, we consider only two possible weather states: hot and cold. Suppose that if today is hot, then the probability that tomorrow is also hot is 0.7, and that if today is cold, the probability that tomorrow is also cold is 0.4. By assigning "hot" to the 0th row and column, and "cold" to the 1st row and column, this Markov chain has the following transition matrix.

|  | hot today | cold today |
|---|---|---|
| hot tomorrow | 0.7 | 0.6 |
| cold tomorrow | 0.3 | 0.4 |

The 0th column of the matrix says that if it is hot today, there is a 70% chance that tomorrow will be hot (0th row) and a 30% chance that tomorrow will be cold (1st row). The 1st column says if it is cold today, then there is a 60% chance of heat and a 40% chance of cold tomorrow.

Markov chains can be represented by a *state diagram*, a type of directed graph. The nodes in the graph are the states, and the edges indicate the state transition probabilities. The Markov chain described above has the following state diagram.




**Problem 1.** Define a `MarkovChain` class whose constructor accepts an $n \times n$ transition matrix $A$ and, optionally, a list of state labels. If $A$ is not column stochastic, raise a `ValueError`. Construct a dictionary mapping the state labels to the row/column index that they correspond to in $A$ (given by order of the labels in the list), and save $A$, the list of labels, and this dictionary as attributes. If there are no state labels given, use the labels $\begin{bmatrix} 0 & 1 & \ldots & n-1 \end{bmatrix}$.

For example, for the weather model described above, the transition matrix is

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix},$$

the list of state labels is `["hot", "cold"]`, and the dictionary mapping labels to indices is `{"hot":0, "cold":1}`. This Markov chain could be also represented by the transition matrix

$$\widetilde{A} = \begin{bmatrix} 0.4 & 0.3 \\ 0.6 & 0.7 \end{bmatrix},$$

the labels `["cold", "hot"]`, and the resulting dictionary `{"cold":0, "hot":1}`.

## Simulating State Transitions

Simulating the weather model described above requires a programmatic way of choosing between the outgoing transition probabilities of each state. For example, if it is cold today, we could flip a weighted coin that lands on tails 60% of the time (guess tomorrow is hot) and heads 40% of the time (guess tomorrow is cold) to predict the weather tomorrow. The *Bernoulli distribution* with parameter $p = 0.4$ simulates this behavior: 60% of draws are 0, and 40% of draws are a 1.

A *binomial distribution* is the sum several Bernoulli draws: one binomial draw with parameters $n$ and $p$ indicates the number of successes out of $n$ independent experiments, each with probability $p$ of success. In other words, $n$ is the number of times to flip the coin, and $p$ is the probability that the coin lands on heads. Thus, a binomial draw with $n = 1$ is a Bernoulli draw.

NumPy does not have a function dedicated to drawing from a Bernoulli distribution; instead, use the more general `np.random.binomial()` with $n = 1$ to make a Bernoulli draw.

```
>>> import numpy as np

# Draw from the Bernoulli distribution with p = .5 (flip one fair coin).
>>> np.random.binomial(n=1, p=.5)
0                              # The coin flip resulted in tails.

# Draw from the Bernoulli distribution with p = .3 (flip one weighted coin).
>>> np.random.binomial(n=1, p=.3)
0                              # Also tails.
```

For the weather model, if the "cold" state corresponds to row and column 1 in the transition matrix, $p$ should be the probability that tomorrow is cold. So, if today is cold, select $p = 0.4$; if today is hot, set $p = 0.3$. Then draw from the binomial distribution with $n = 1$ and the selected $p$. If the result is 0, transition to the "hot" state; if the result is 1, stay in the "cold" state.

Using Bernoulli draws to determine state transitions works for Markov chains with two states, but larger Markov chains require draws from a *categorical distribution*, a multivariate generalization of the Bernoulli distribution. A draw from a categorical distribution with parameters $(p_1, p_2, \ldots, p_k)$ satisfying $\sum_{i=1}^{k} p_i = 1$ indicates which of $k$ outcomes occurs. If $k = 2$, a draw simulates a coin flip (a Bernoulli draw); if $k = 6$, a draw simulates rolling a six-sided die. Just as the Bernoulli distribution is a special case of the binomial distribution, the categorical distribution is a special case of the *multinomial distribution* which indicates how many times each of the $k$ outcomes occurs in $n$ repeated experiments. Use `np.random.multinomial()` with $n = 1$ to make a categorical draw.

```
# Draw from the categorical distribution (roll a fair four-sided die).
>>> np.random.multinomial(1, np.array([1./4, 1./4, 1./4, 1./4]))
array([0, 0, 0, 1])     # The roll resulted in a 3.

# Draw from another categorical distribution (roll a weighted four-sided die).
>>> np.random.multinomial(1, np.array([.5, .3, .2, 0]))
array([0, 1, 0, 0])     # The roll resulted in a 1.
```

Consider a four-state weather model with the transition matrix

$$\begin{array}{c c} & \begin{array}{cccc} \text{hot} & \text{mild} & \text{cold} & \text{freezing} \end{array} \\ \begin{array}{c} \text{hot} \\ \text{mild} \\ \text{cold} \\ \text{freezing} \end{array} & \begin{bmatrix} 0.5 & 0.3 & 0.1 & 0 \\ 0.3 & 0.3 & 0.3 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 \\ 0 & 0.1 & 0.2 & 0.2 \end{bmatrix} \end{array}.$$

If today is hot, the probabilities of transitioning to each state are given by the "hot" column of the transition matrix. Therefore, to choose a new state, draw from the categorical distribution with parameters $(0.5, 0.3, 0.2, 0)$. The result $\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$ indicates a transition to the state corresponding to the 1st row and column (tomorrow is mild), while the result $\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$ indicates a transition to the state corresponding to the 2nd row and column (tomorrow is cold). In other words, the position of the 1 tells which column of the matrix to use as the parameters for the next categorical draw.

**Problem 2.** Write a method for the `MarkovChain` class that accepts a single state label. Use the label-to-index dictionary to determine the column of $A$ that corresponds to the provided state label, then draw from the corresponding categorical distribution to choose a state to transition to. Return the corresponding label of the new state (not its index).
(Hint: `np.argmax()` may be useful.)

**Problem 3.** Add the following methods to the `MarkovChain` class.

- `walk()`: Accept a state label and an interger $N$. Starting at the specified state, use your method from Problem 2 to transition from state to state $N-1$ times, recording the state label at each step. Return the list of $N$ state labels, including the initial state.
- `path()`: Accept labels for an initial state and an end state. Beginning at the initial state, transition from state to state until arriving at the specified end state, recording the state label at each step. Return the list of state labels, including the initial and final states.

Test your methods on the two-state and four-state weather models described previously.

## General State Distributions

For a Markov chain with $n$ states, the probability of being in each state can be encoded by a $n$-vector $\mathbf{x}$, called a *state distribution vector*. The entries of $\mathbf{x}$ must be nonnegative and sum to 1, and the $i$th entry $x_i$ of $\mathbf{x}$ is the probability of being in state $i$. For example, the state distribution vector $\mathbf{x} = \begin{bmatrix}0.8 & 0.2\end{bmatrix}^\mathsf{T}$ corresponding to the 2-state weather model indicates an 80% chance that today is hot and a 20% chance that today is cold. On the other hand, the vector $\mathbf{x} = \begin{bmatrix}0 & 1\end{bmatrix}^\mathsf{T}$ implies that today is, with 100% certainty, cold.

If $A$ is a transition matrix for a Markov chain with $n$ states and $\mathbf{x}$ is a corresponding state distribution vector, then $A\mathbf{x}$ is also a state distribution vector. In fact, if $\mathbf{x}_k$ is the state distribution vector corresponding to a certain time $k$, then $\mathbf{x}_{k+1} = A\mathbf{x}_k$ contains the probabilities of being in each state after allowing the system to transition again. For the weather model, this means that if there is an 80% chance that it will be hot 5 days from now, written $\mathbf{x}_5 = \begin{bmatrix}0.8 & 0.2\end{bmatrix}^\mathsf{T}$, then since

$$\mathbf{x}_6 = A\mathbf{x}_5 = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix} \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.68 \\ 0.32 \end{bmatrix},$$

there is a 68% chance that 6 days from now will be a hot day.

### Convergent Transition Matrices

Given an initial state distribution vector $\mathbf{x}_0$, defining $\mathbf{x}_{k+1} = A\mathbf{x}_k$ yields the significant relation

$$\mathbf{x}_k = A\mathbf{x}_{k-1} = A(A\mathbf{x}_{k-2}) = A(A(A\mathbf{x}_{x-3})) = \cdots = A^k\mathbf{x}_0.$$

This indicates that the $(i,j)$th entry of $A^k$ is the probability of transition from state $j$ to state $i$ in $k$ steps. For the transition matrix of the 2-state weather model, a pattern emerges in $A^k$ for even

small values of $k$:

$$A = \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0.67 & 0.66 \\ 0.33 & 0.34 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0.667 & 0.666 \\ 0.333 & 0.334 \end{bmatrix}.$$

As $k \to \infty$, the entries of $A^k$ converge, written

$$\lim_{k\to\infty} A^k = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix}. \tag{1.1}$$

In addition, for any initial state distribution vector $\mathbf{x}_0 = [a,\ b]^\mathsf{T}$ (meaning $a, b \geq 0$ and $a + b = 1$),

$$\lim_{k\to\infty} \mathbf{x}_k = \lim_{k\to\infty} A^k \mathbf{x}_0 = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 2(a+b)/3 \\ (a+b)/3 \end{bmatrix} = \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix}.$$

Thus, $\mathbf{x}_k \to \mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ as $k \to \infty$, regardless of the initial state distribution $\mathbf{x}_0$. So, according to this model, no matter the weather today, the probability that it is hot a week from now is approximately 66.67%. In fact, approximately 2 out of 3 days in the year should be hot.

## Steady State Distributions

The state distribution $\mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ has another important property:

$$A\mathbf{x} = \begin{bmatrix} 7/10 & 3/5 \\ 3/10 & 2/5 \end{bmatrix} \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 14/30 + 3/15 \\ 6/30 + 2/15 \end{bmatrix} = \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix} = \mathbf{x}.$$

Any $\mathbf{x}$ satisfying $A\mathbf{x} = \mathbf{x}$ is called a *steady state distribution* or a *stable fixed point* of $A$. In other words, a steady state distribution is an eigenvector of $A$ corresponding to the eigenvalue $\lambda = 1$.

Every finite Markov chain has at least one steady state distribution. If some power $A^k$ of $A$ has all positive (nonzero) entries, then the steady state distribution is unique.[1] In this case, $\lim_{k\to\infty} A^k$ is the matrix whose columns are all equal to the unique steady state distribution, as in (1.1). Under these circumstances, the steady state distribution $\mathbf{x}$ can be found by iteratively calculating $\mathbf{x}_{k+1} = A\mathbf{x}_k$, as long as the initial vector $\mathbf{x}_0$ is a state distribution vector.

**Achtung!**

Though every Markov chain has at least one steady state distribution, the procedure described above fails if $A^k$ fails to converge. For instance, consider the transition matrix

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad A^k = \begin{cases} A & \text{if } k \text{ is odd} \\ I & \text{if } k \text{ is even.} \end{cases}$$

In this case as $k \to \infty$, $A^k$ oscillates between two different matrices.

Furthermore, the steady state distribution is not always unique; the transition matrix defined above, for example, has infinitely many.

[1] This is a consequence of the *Perron-Frobenius theorem*, which is presented in detail in Volume 1.

**Problem 4.** Write a method for the MarkovChain class that accepts a convergence tolerance tol and a maximum number of iterations maxiter. Generate a random state distribution vector $\mathbf{x}_0$ and calculate $\mathbf{x}_{k+1} = A\mathbf{x}_k$ until $\|\mathbf{x}_{k-1} - \mathbf{x}_k\|_1 < \texttt{tol}$, where $A$ is the transition matrix saved in the constructor. If $k$ exceeds maxiter, raise a ValueError to indicate that $A^k$ does not converge. Return the approximate steady state distribution $\mathbf{x}$ of $A$.

To test your function, generate a random transition matrix $A$. Verify that $A\mathbf{x} = \mathbf{x}$ and that the columns of $A^k$ approach $\mathbf{x}$ as $k \to \infty$. To compute $A^k$, use NumPy's (very efficient) algorithm for computing matrix powers.

```
>>> A = np.array([[.7, .6],[.3, .4]])
>>> np.linalg.matrix_power(A, 10)       # Compute A^10.
array([[ 0.66666667,  0.66666667],
       [ 0.33333333,  0.33333333]])
```

Finally, use your method to validate the results of Problem 3: for the two-state and four-state weather models,

1. Calculate the steady state distribution corresponding to the transition matrix.
2. Run a weather simulation for a large number of days using walk() and verify that the results match the steady state distribution (for example, approximately 2/3 of the days should be hot for the two-state model).

**Note**

Problem 4 is a special case of the *power method*, an algorithm for calculating an eigenvector of a matrix corresponding to the eigenvalue of largest magnitude. The general power method, together with a discussion of its convergence conditions, is discussed in Volume 1.

## Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study *natural languages*, meaning spoken or written languages like English or Russian [?]. In the early 20th century, Markov used his chains to model how Russian switched from vowels to consonants. By mid-century, they had been used as an attempt to model English. It turns out that plain Markov chains are, by themselves, insufficient to model or produce very good English. However, they can approach a fairly good model of bad English, with sometimes amusing results.

By nature, a Markov chain is only concerned with its current state, not with previous states. A Markov chain simulating transitions between English words is therefore completely unaware of context or even of previous words in a sentence. For example, if a chain's current state is the word "continuous," the chain may say that the next word in a sentence is more likely to be "function" rather than "raccoon." However the phrase "continuous function" may be gibberish in the context of the rest of the sentence.

## Generating Random Sentences

Consider the problem of generating English sentences that are similar to the text contained in a specific file, called the *training set*. The goal is to construct a Markov chain whose states and transition probabilities represent the vocabulary and—hopefully—the style of the source material. There are several ways to approach this problem, but one simple strategy is to assign each unique word in the training set to a state, then construct the transition probabilities between the states based on the ordering of the words in the training set. To indicate the beginning and end of a sentence requires two extra states: a *start state*, $start, marking the beginning of a sentence; and a *stop state*, $top, marking the end. The start state should only transitions to words that appear at the beginning of a sentence in the training set, and only words that appear at the end a sentence in the training set should transition to the stop state.

Consider the following small training set, paraphrased from Dr. Seuss [**?**].

```
I am Sam Sam I am.
Do you like green eggs and ham?
I do not like them, Sam I am.
I do not like green eggs and ham.
```

There are 15 unique words in this training set, including punctuation (so "ham?" and "ham." are counted as distinct words) and capitalization (so "Do" and "do" are also different):

I am Sam am. Do you like green
eggs and ham? do not them, ham.

With start and stop states, the transition matrix should be $17 \times 17$. Each state must be assigned a row and column index in the transition matrix, for example,

| $start | I | am | Sam | ... | ham. | $top |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | ... | 15 | 16 |

The $(i, j)$th entry of the transition matrix $A$ should be the probability that word $j$ is followed by word $i$. For instance, the word "Sam" is followed by the words "Sam" once and "I" twice in the training set, so the state corresponding to "Sam" (index 3) should transition to the state for "Sam" with probability $1/3$, and to the state for "I" (index 1) with probability $2/3$. That is, $A_{3,3} = 1/3$, $A_{1,3} = 2/3$, and $A_{i,3} = 0$ for $i \notin \{1, 3\}$. Similarly, the start state should transition to the state for "I" with probability $3/4$, and to the state for "Do" with probability $1/4$; the states for "am.", "ham?", and "ham." should each transition to the stop state.

To construct the transition matrix, parse the training set and add 1 to $A_{i,j}$ every time word $j$ is followed by word $i$, in this case arriving at the matrix

$$
\begin{array}{c} \\ \text{\$start} \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \\ \text{ham.} \\ \text{\$top} \end{array}
\begin{array}{c} \begin{array}{ccccccc} \text{\$start} & \text{I} & \text{am} & \text{Sam} & & \text{ham.} & \text{\$top} \end{array} \\
\left[\begin{array}{ccccccc}
0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
3 & 0 & 0 & 2 & \cdots & 0 & 0 \\
0 & 1 & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 1 & 0
\end{array}\right] \end{array}.
$$

To avoid a column of zeros, set $A_{j,j} = 1$ where j is the index of the stop state (so the stop state always transitions to itself). Next, divide each column by its sum so that each column sums to 1:

$$\begin{array}{c} \text{\$tart} \\ \text{I} \\ \text{am} \\ \text{Sam} \\ \\ \text{ham.} \\ \text{\$top} \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 3/4 & 0 & 0 & 2/3 & \dots & 0 & 0 \\ 0 & 1/5 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 1/3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 \end{bmatrix}.$$

(Column labels: \$tart, I, am, Sam, …, ham., \$top)

The 3/4 indicates that 3 out of 4 times, the sentences in the training set start with the word "I". Similarly, the 2/3 and 1/3 says that "Sam" is followed by "I" twice and by "Sam" once in the training set. Note that "am" (without a period) always transitions to "Sam" and that "ham." (with a period) always transitions the stop state.

The entire procedure of creating the transition matrix for the Markov chain with words from a file as states is summarized below.

**Algorithm 1.1** Convert a training set of sentences into a Markov chain.

```
1: procedure MakeTransitionMatrix(filename)
2:     Read the training set from the file filename.
3:     Get the set of unique words in the training set (the state labels).
4:     Add labels "$tart" and "$top" to the set of states labels.
5:     Initialize an appropriately sized square array of zeros to be the transition matrix.
6:     for each sentence in the training set do
7:         Split the sentence into a list of words.
8:         Prepend "$tart" and append "$top" to the list of words.
9:         for each consecutive pair (x, y) of words in the list of words do
10:            Add 1 to the entry of the transition matrix that corresponds to
                   transitioning from state x to state y.
11:    Make sure the stop state transitions to itself.
12:    Normalize each column by dividing by the column sums.
```

**Problem 5.** Write a class called `SentenceGenerator` that inherits from the `MarkovChain` class. The constructor should accept a filename (the training set). Read the file and build a transition matrix from its contents as described in Algorithm 1.1. Save the same attributes as the constructor of `MarkovChain` does so that inherited methods work correctly. Assume that the training set has one complete sentence written on each line.
(Hint: if the contents of the file are in the string `s`, then `s.split()` is the list of words and `s.split('n')` is the list of sentences.)

### Note

The Markov chains that result from the procedure in Problem 5 have a few interesting structural characteristics. The stop state is a *sink*, meaning it only transitions to itself. Because of this, and since every node has a path to the stop state, any traversal of the chain will end up in the stop state forever. The stop state is therefore called an *absorbing state*, and the chain as a whole is called an *absorbing Markov chain*. Furthermore, the steady state is the vector with a 1 in the entry corresponding to the stop state and 0s everywhere else.

**Problem 6.** Add a method to the `SentenceGenerator` class called `babble()`. Use the `path()` method from Problem 3 to generate a random sentence based on the training document. That is, generate a path from the start state to the stop state, remove the `"$tart"` and `"$top"` labels from the path, and join the resulting list together into a single, space-separated string.

For example, your `SentenceGenerator` class should be able to create random sentences that sound somewhat like Yoda speaking.

```
>>> yoda = SentenceGenerator("yoda.txt")
>>> for _ in range(3):
...     print(yoda.babble())
...
Impossible to my size, do not!
For eight hundred years old to enter the dark side of Congress there is.
But beware of the Wookiees, I have.
```

# Additional Material

## Other Applications of Markov Chains

Markov chains are a useful way to study many probabilistic phenomena, so they have a wide variety of applications. The following are just a few that are covered in other parts of this lab manual series.

- **PageRank**: Google's PageRank algorithm uses a Markov chain-based approach to rank web pages. The main idea is to use the entries of the steady state vector as a measure of importance for the corresponding states. For example, the steady state $\mathbf{x} = \begin{bmatrix} 2/3 & 1/3 \end{bmatrix}^\mathsf{T}$ for the two-state weather model means that the hot state is "more important" (occurs more frequently) than the cold state. See the PageRank lab in Volume 1.
- **MCMC Sampling**: A *Monte Carlo Markov Chain* (MCMC) method constructs a Markov chain whose steady state is a probability distribution that is difficult to sample from directly. This provides a way to sample from nontrivial or abstract distributions. Many MCMC methods are used in various fields, from machine learning to physics. See the Volume 3 lab on the Metropolis-Hastings algorithm.
- **Hidden Markov Models**: The Markov chain simulations in this lab use an initial condition (a state distribution vector $\mathbf{x}_0$) and known transition probabilities to make predictions forward in time. Converseyl, a *hidden Markov model* (HMM) assumes that a given set of observations are the result of a Markov process, then uses those observations to infer the corresponding transition probabilities. Hidden Markov models are used extensively in modern machine learning, especially for speech and language processing. See the Volume 3 lab on Speech Recognition.

## Large Training Sets

The approach in Problems 5 and 6 begins to fail as the training set grows larger. For example, a single Shakespearean play may not be large enough to cause memory problems, but *The Complete Works of William Shakespeare* certainly will.

To accommodate larger data sets, consider use a sparse matrix from `scipy.sparse` for the transition matrix instead of a regular NumPy array. Specifically, construct the transition matrix as a `lil_matrix` (which is easy to build incrementally), then convert it to the `csc_matrix` format (which supports fast column operations). Ensure that the process still works on small training sets, then proceed to larger training sets. How are the resulting sentences different if a very large training set is used instead of a small training set?

## Variations on the English Model

Choosing a different state space for the English Markov model produces different results. Consider modifying the `SentenceGenerator` class so that it can determine the state space in a few different ways. The following ideas are just a few possibilities.

- Let each punctuation mark have its own state. In the Dr. Seuss training set, instead of having two states for the words "ham?" and "ham.", there would be three states: "ham", "?", and ".", with "ham" transitioning to both punctuation states.
- Model paragraphs instead of sentences. Add a $tartParagraph state that always transitions to $tartSentence and a $topParagraph state that is sometimes transitioned to by $topSentence.

- Let the states be individual letters instead of individual words. Be sure to include a state for the spaces between words.
- Construct the state space so that the next state depends on both the current and previous states. This kind of Markov chain is called a *Markov chain of order 2*. This way, every set of three consecutive words in a randomly generated sentence should be part of the training set, as opposed to only every consecutive pair of words coming from the set.
- Instead of generating random sentences from a single source, simulate a random conversation between $n$ people. Construct a Markov chain $M_i$, for each person, $i = 1, \ldots, n$, then create a Markov chain $C$ describing the conversation transitions from person to person; in other words, the states of $C$ are the $M_i$. To create the conversation, generate a random sentence from the first person using $M_1$. Then use $C$ to determine the next speaker, generate a random sentence using their Markov chain, and so on.

## Natural Language Processing Tools

The Markov model of Problems 5 and 6 is a *natural language processing* application. Python's `nltk` module (natural language toolkit) has many tools for parsing and analyzing text for these kinds of problems [**?**]. For example, `nltk.sent_tokenize()` reads a single string and splits it up into sentences. This could be useful, for example, in making the `SentenceGenerator` class compatible with files that do not have one sentence per line.

```
>>> from nltk import sent_tokenize
>>> with open("yoda.txt", 'r') as yoda:
...     sentences = sent_tokenize(yoda.read())
...
>>> print(sentences)
['Away with your weapon!',
 'I mean you no harm.',
 'I am wondering - why are you here?',
 ...
```

The `nltk` module is **not** part of the Python standard library. For instructions on downloading, installing, and using `nltk`, visit `http://www.nltk.org/`.