5 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 apply a Markov chain to a natural language processing problem.*

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 die 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.

Finite Markov chains have an associated *transition matrix* that stores the information about the 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.

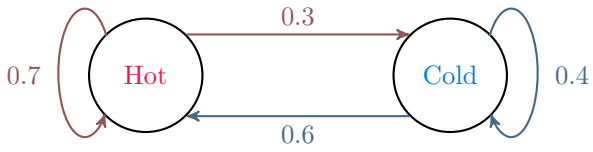
> **Note**
>
> A transition matrix where the columns sum to 1 is called *column stochastic* (or *left stochastic*). The rows of a *row stochastic* (or *right stochastic*) transition matrix each sum to 1 and the (i, j)th entry of the matrix is the probability of moving from state i to state j. Both representations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

Consider a very simple weather model where the probability of being hot or cold depends on the weather of the previous day. If the probability that tomorrow is hot given that today is hot is 0.7, and the probability that tomorrow is cold given that today is cold is 0.4, then by assigning hot to the 0th row and column, and cold to the 1^{st} row and column, this Markov chain has the following transition matrix.

$$\begin{array}{r} \\ \text{hot tomorrow} \\ \text{cold tomorrow} \end{array} \begin{array}{c} \begin{array}{cc} \text{hot today} & \text{cold today} \end{array} \\ \begin{bmatrix} 0.7 & 0.6 \\ 0.3 & 0.4 \end{bmatrix} \end{array}$$

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. Transition matrices for Markov chains are efficiently stored as NumPy arrays. Write a function that accepts an integer n and returns the transition matrix for a random Markov chain with n states.
(Hint: use array broadcasting to avoid looping.)

Simulating State Transitions

A single draw from a *binomial distribution* with parameters n and p indicates the number of successes out of n independent experiments, each with probability p of success. The classic example is a series of coin flips, where p is the probability that the coin lands heads side up. NumPy's `random` module has an efficient tool, `binomial()`, for drawing from a binomial distribution.

```
>>> import numpy as np

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

Consider again the simple weather model and suppose that today is hot. The column that corresponds to "hot"in the transition matrix is $[0.7, 0.3]$. To determine whether tomorrow is hot or cold, draw from the binomial distribution with $n = 1$ and $p = 0.3$. If the draw is 1, which has 30% likelihood, then tomorrow is cold. If the draw is 0, which has 70% likelihood, then tomorrow is hot. The following function implements this idea.

```
def forecast():
    """Forecast tomorrow's weather given that today is hot."""
    transition = np.array([[0.7, 0.6], [0.3, 0.4]])
```

```
    # Sample from a binomial distribution to choose a new state.
    return np.binomial(1, transition[1, 0])
```

Problem 2. Modify `forecast()` so that it accepts an integer parameter `days` and runs a simulation of the weather for the number of days given. Return a list containing the day-by-day weather predictions (0 for hot, 1 for cold). Assume the first day is hot, but do not include the data from the first day in the list of predictions. The resulting list should therefore have `days` entries.

Larger Chains

The `forecast()` function makes one random draw from a binomial distribution to simulate a state change. Larger Markov chains require draws from a *multinomial distribution*, a multivariate generalization of the binomial distribution. A draw from a multinomial distribution parameters n and $(p_1,\ p_2,\ \ldots,\ p_k)$ indicates which of k outcomes occurs in n different experiments. In this case the classic example is a series of dice rolls, with 6 possible outcomes of equal probability.

```
>>> die_probabilities = np.array([1./6, 1./6, 1./6, 1./6, 1./6, 1./6])

# Draw from the multinomial distribution with n = 1 (roll a single die).
>>> np.random.multinomial(1, die_probabilities)
array([0, 0, 0, 1, 0, 0])     # The roll resulted in a 4.
```

Problem 3. Let the following matrix be the transition matrix for a Markov chain modeling weather with four states: hot, mild, cold, and freezing.

	hot	mild	cold	freezing
hot	0.5	0.3	0.1	0
mild	0.3	0.3	0.3	0.3
cold	0.2	0.3	0.4	0.5
freezing	0	0.1	0.2	0.2

Write a new function that accepts an integer parameter and runs the same kind of simulation as `forecast()`, but that uses this new four-state transition matrix. This time, assume that the first day is mild. Return a list containing the day-to-day results (0 for hot, 1 for mild, 2 for cold, and 3 for freezing).

General State Distributions

For a Markov chain with n states, the probability of being in each of the states can be encoded by a single $n \times 1$ vector $\mathbf{x}$, called a *state distribution vector*. The entries of $\mathbf{x}$ must be nonnegative and sum to 1. Then the ith entry x_i of $\mathbf{x}$ is the probability of being in state i. For example, the state

distribution vector $\mathbf{x} = [0.8,\ 0.2]^\mathsf{T}$ corresponding to the 2-state weather model of Problem 2 indicates that there is a 80% chance that today is hot and a 20% chance that today is cold. On the other hand, the vector $\mathbf{x} = [0, 1]^\mathsf{T}$ implies that today is, with 100% certainty, cold.

If A is an $n \times n$ transition matrix for a Markov chain and $\mathbf{x}$ is a 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 $\mathbf{x}_5 = [0.8,\ 0.2]^\mathsf{T}$, 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 following 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, something curious happens to 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 as follows.

$$\lim_{k\to\infty} A^k = \begin{bmatrix} 2/3 & 2/3 \\ 1/3 & 1/3 \end{bmatrix}. \tag{5.1}$$

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

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

Thus as $k \to \infty$, $\mathbf{x}_k \to \mathbf{x} = [2/3,\ 1/3]^\mathsf{T}$, 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 $\mathbf{x} = [2/3,\ 1/3]^\mathsf{T}$ 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 with corresponding eigenvalue $\lambda = 1$.

Every 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 (5.1). 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. Consider the following example.

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

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.

Problem 4. Write a function that accepts an $n \times n$ transition matrix A, a convergence tolerance ϵ, and a maximum number of iterations N. Generate a random state distribution vector $\mathbf{x}_0$ and calculate $\mathbf{x}_{k+1} = A\mathbf{x}_k$ until $\|\mathbf{x}_{k-1} - \mathbf{x}_k\| < \epsilon$. If k exceeds N, 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, use Problem 1 to 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 function to validate the results of Problems 2 and 3:

1. Calculate the steady state distributions corresponding to the transition matrices for each simulation.
2. Run each simulation for a large number of days and verify that the results match the steady state distribution (for example, check that approximately 2/3 of the days are hot for the smaller weather model).

[1] This is a consequence of the *Perron-Frobenius theorem*, which is presented in conjunction with spectral calculus in Volume I.

> **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 version of the power method, together with a discussion of its convergence conditions, is discussed in another lab.

Using Markov Chains to Simulate English

One of the original applications of Markov chains was to study natural languages. 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 Markov chains are, by themselves, insufficient to model 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. Thus a Markov chain simulating transitions between English words is completely unaware of context or even of previous words in a sentence. For example, a Markov chain's current state may be the word "continuous." Then the chain would say that the next word in the sentence is more likely to be "function" rather than "raccoon." However, without the context of the rest of the sentence, even two likely words stringed together may result in gibberish.

We restrict ourselves to a subproblem of modeling the English of a specific file. The transition probabilities of the resulting Markov chain reflect the sort of English that the source authors speak. Thus the Markov chain built from *The Complete Works of William Shakespeare* differs greatly from, say, the Markov chain built from a collection of academic journals. We call the source collection of works in the next problems the *training set*.

Making the Chain

With the weather models of the previous sections, we chose a fixed number of days to simulate. However, English sentences are of varying length, so we do not know beforehand how many words to choose (how many state transitions to make) before ending the sentence. To capture this feature, we include two extra states in our Markov model: a *start state* (\$tart) marking the beginning of a sentence, and a *stop state* (\$top) marking the end. Thus a training set with N unique words has an $(N+2) \times (N+2)$ transition matrix.

The start state 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 transition to the stop state. The stop state is called an *absorbing state* because once we reach it, we cannot transition back to another state.

After determining the states in the Markov chain, we need to determine the transition probabilities between the states and build the corresponding transition matrix. Consider the following small training set from Dr. Seuss as an example.

```
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.
```

If we include punctuation (so "ham?" and "ham." are counted as distinct words) and do not alter the capitalization (so "Do" and "do" are also different), there are 15 unique words in this training set:

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 17×17. Each state must be assigned a row and column index in the transition matrix. As easy way to do this is to assign the states an index based on the order that they appear in the training set. Thus our states and the corresponding indices will be as follows:

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

The start state should transition to the words "I" and "Do", and the words "am.", "ham?", and "ham." should each transition to the stop state. We first count the number of times that each state transitions to another state:

	\$start	I	am	Sam		ham.	\$top
\$start	0	0	0	0	...	0	0
I	3	0	0	2	...	0	0
am	0	1	0	0	...	0	0
Sam	0	0	1	1	...	0	0
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
ham.	0	0	0	0	...	0	0
\$top	0	0	0	0	...	1	1

Now divide each column by its sum so that each column sums to 1.

	\$start	I	am	Sam		ham.	\$top
\$start	0	0	0	0	...	0	0
I	3/4	0	0	2/3	...	0	0
am	0	1/5	0	0	...	0	0
Sam	0	0	1	1/3	...	0	0
	⋮	⋮	⋮	⋮	⋱	⋮	⋮
ham.	0	0	0	0	...	0	0
\$top	0	0	0	0	...	1	1

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 tell us 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. Finally, to avoid a column of zeros, we place a 1 in the bottom right hand corner of the matrix (so the stop state always transitions to itself).

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

Algorithm 5.1 Convert a training set of sentences into a Markov chain.

```
procedure MakeTransitionMatrix
    Count the number of unique words in the training set.
    Initialize a square array of zeros of the appropriate size to be the transition
        matrix (remember to account for the start and stop states).
    Initialize a list of states, beginning with "$tart".
    for each sentence in the training set do
        Split the sentence into a list of words.
        Add each new word in the sentence to the list of states.
        Convert the list of words into a list of indices indicating which row and
            column of the transition matrix each word corresponds to.
        Add 1 to the entry of the transition matrix corresponding to
            transitioning from the start state to the first word of the sentence.
        for each consecutive pair (x, y) of words in the list of words do
            Add 1 to the entry of the transition matrix corresponding to
                transitioning from state x to state y.
        Add 1 to the entry of the transition matrix corresponding to
            transitioning from the last word of the sentence to the stop state.
    Make sure the stop state transitions to itself.
    Normalize each column by dividing by the column sums.
```

Problem 5. Write a class called `SentenceGenerator`. The constructor should accept a filename (the training set). Read the file and build a transition matrix from its contents as described in Algorithm 5.1.

You may assume that the file has one complete sentence written on each line, and your implementation may be either column- or row-stochastic.

Problem 6. Add a method to the `SentenceGenerator` class called `babble()`. Begin at the start state and use the strategy from Problem 3 to repeatedly transition through the object's Markov chain. Keep track of the path through the chain and the corresponding sequence of words. When the stop state is reached, stop transitioning to terminate the simulation. Return the resulting sentence as a single 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

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 your `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 example 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 from $topSentence.
- Let the states be individual letters instead of individual words. Be sure to include a state for the spaces between words. We will explore this particular state space choice more in Volume III together with hidden Markov models.
- 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.

```
>>> 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/`.