Markov models; numpy

Ben Bolker

31 October 2019

## Markov models

* In a **Markov model**, the future state of a system depends only on its current state (not on any previous states)
* Widely used: physics, chemistry, queuing theory, economics, genetics, mathematical biology, sports, …
* From the [Markov chain page on Wikipedia](https://en.wikipedia.org/wiki/Markov_chain):
	+ Suppose that you start with $10, and you wager $1 on an unending, fair, coin toss indefinitely, or until you lose all of your money. If $X\_{n}$ represents the number of dollars you have after $n$ tosses, with $X\_{0}=10$, then the sequence $\{X\_{n}:n\in N\}$ is a Markov process.
	+ If I know that you have $12 now, then you will either have $11 or $13 after the next toss with equal probability
	+ Knowing the history (that you started with $10, then went up to $11, down to $10, up to $11, and then to $12) doesn’t provide any more information

## Markov models for text analysis

* A Markov model of text would say that the *next* word in a piece of text (or letter, depending on what scale we’re working at) depends only on the *current* word
* We will write a program to analyse some text and, based on the frequency of word pairs, produce a short “sentence” from the words in the text, using the Markov model

## Issues

* The text that we use, for example Kafka’s *Metamorphosis* (<http://www.gutenberg.org/files/5200/5200.txt>) or Melville’s *Moby Dick* (<http://www.gutenberg.org/files/2701/2701-0.txt>), will contain lots of symbols, such as punctuation, that we should remove first
* It’s easier if we convert all words to lower case
* The text that we use will either be in a file stored locally, or maybe accessed using its URL.
* There is a random element to Markov processes and so we will need to be able to generate numbers randomly (or pseudo-randomly)

## Cleaning strings

* text/data cleaning is an inevitable part of dealing with text files or data sets.
* We can use the .lower() method to convert all upper case letters to lower case
* python has a function called translate() that can be used to scrub certain characters from a string, but it is a little complicated (see <https://machinelearningmastery.com/clean-text-machine-learning-python/>)

## text cleaning example

* A function to delete from a given string s the characters that appear in the string delete\_chars.
* Python has a built-in string string.punctuation:

import string
print(string.punctuation)

## !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

def clean\_string(s,delete\_chars=string.punctuation):
 for i in delete\_chars:
 s = s.replace(i,"")
 return(s)
x = "ab,Cde!?Q@#$I"
print(clean\_string(x))

## abCdeQI

## Markov text model algorithm

1. Open and read the text file.
2. Clean the file.
3. Create the text dictionary with each word as a key and the words that come next in the text as a list.
4. Randomly select a starting word from the text and then create a “sentence” of a specified length using randomly selected words from the dictionary

## markov\_create function (outline)

def markov\_create(file\_name, sentence\_length = 20):
 ## open the file and store its contents in a string
 text\_file = open(file\_name, 'r')
 text = text\_file.read()
 ## clean the text and then split it into words
 clean\_text = clean\_string(text)
 word\_list = clean\_text.split()
 ## create the markov dictionary
 text\_dict = markov\_dict(word\_list)
 ## Produce a sentence (a list of strings) of length
 ## sentence\_length using the dictionary
 sentence = markov\_sentence(text\_dict, sentence\_length)
 ## print out the sentence as a string using
 ## the .join() method.
 return " ".join(sentence)

## the rest of it

To complete this exercise, we need to produce the following functions:

* clean\_string(s,delete\_chars = string.punctuation) strips the text of punctuation and converts upper case words into lower case.
* markov\_dict(word\_list) creates a dictionary from a list of words
* markov\_sentence(text\_dict, sentence\_length) randomly produces a sentence using the dictionary.

## the random module

* The random module can be used to generate pseudo-random numbers or to pseudo-randomly select items.
* docs: <https://docs.python.org/3/library/random.html>
* randrange() picks a random integer from a prescribed range can be generated
* choice(seq) randomly chooses an element from a sequence, such as a list or tuple
* shuffle shuffles (permutes) the items in a list; sample() samples elements from a list, tuple, or set
* random.seed() sets the starting value for a (pseudo-)random number sequence [**important**]

## random examples

import random
random.seed(101) ## any integer you want
random.randrange(2, 102, 2) # random even integers

## 76

random.choice([1, 2, 3, 4, 5]) # random choice from list
## random.choices([1, 2, 3, 4, 5], 9) # multiple choices (Python >=3.6)

## 2

random.sample([1, 2, 3, 4, 5], 3) # rand. sample of 3 items

## [5, 3, 2]

random.random() # uniform random float between 0 and 1

## 0.048520987208713895

random.uniform(3, 7) # uniform random between 3 and 7

## 5.014081424907534

## why random-number seeds?

* start from the same point every time
* for **reproducibility** and **debugging**
	+ across computers
	+ across operating systems
	+ across sessions
* set seed at the beginning of each session/notebook

random.seed(101)
for i in range(3):
 print(random.randrange(10))

## 9
## 3
## 8

random.seed(101)
for i in range(3):
 print(random.randrange(10))

## 9
## 3
## 8

## numpy Installation

numpy is the fundamental package for scientific computing with Python. It contains among other things:

* a powerful N-dimensional array object
* **broadcasting** to run a function across rows/columns
* linear algebra and random number capabilities

numpy should already be installed with Anaconda or on syzygy. If not, you Good documentation can be found [here](https://docs.scipy.org/doc/numpy/user/) and [here](http://www.engr.ucsb.edu/~shell/che210d/numpy.pdf).

## arrays

* The array() is numpy’s main data structure.
* Similar to a Python list, but must be *homogeneous* (e.g. floating point (float64) or integer (int64) or str)
* numpy is also more precise about numeric types (e.g. float64 is a *64-bit* floating point number)

## array examples

import numpy as np ## use "as np" so we can abbreviate
x = [1, 2, 3]
a = np.array([1, 4, 5, 8], dtype=float)
print(a)

## [1. 4. 5. 8.]

print(type(a))

## <class 'numpy.ndarray'>

print(a.shape)

## (4,)

## shape

* the shape of an array is a tuple that lists its dimensions
* np.array([1,2]) produces a 1-dimensional (1-D) array of length 2 whose entries have type int
* np.array([1,2], float) produces a 1-dimensional (1-D) array of length 2 whose entries have type float64.

a1 = np.array([1,2])
print(a1.dtype)

## int64

print(a1.shape)

## (2,)

print(len(a1))

## 2

a2 = np.array([1,2],float)
print(a2.dtype)

## float64

* arrays can be created from lists or tuples.
* arrays can also be created using the range function.
* numpy has a function called np.arange (like range) that creates arrays
* np.zeros() and np.ones() create arrays of all zeros or all ones

## more array examples

x = [1, 'a', 3]
a = np.array(x) ## what happens?
b = np.array(range(10), float)
c = np.arange(5, dtype=float)
d = np.arange(2,4, 0.5, dtype=float)
np.ones(10)

## array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

np.zeros(4)

## array([0., 0., 0., 0.])

## slicing and indexing

* slicing and indexing of 1-D arrays works the same way as lists/tuples/strings
* arrays are *mutable* like lists/dictionaries, so we can set elements (e.g. a[1]=0)
* or use the .copy() method to make a new, independent copy (works for lists etc. too!)

## slicing/indexing examples

a1 = np.array([1.0, 2, 3, 4, 5, 6])
a1[1]

## 2.0

a1[:-3]

## array([1., 2., 3.])

b1 = a1
c1 = a1.copy()
b1[1] = 23
a1[1]

## 23.0

c1[1]

## 2.0

## Multi-dimensional arrays

* We have used nested lists of lists to represent matrices.
* numpy’s 2-dimensional arrays serve the same purpose but are (much) easier to work with
* they can be created by passing a list of lists/tuple of tuples to the np.array() function
* **Elements of an array are indexed via a[i,j] rather than a[i][j]**

## examples

nested = [[1, 2, 3], [4, 5, 6]]
a = np.array(nested, float)
nested[0][2]

## 3

a[0,2]

## 3.0

a

## array([[1., 2., 3.],
## [4., 5., 6.]])

a.shape

## (2, 3)

## slicing and reshaping multi-dimensional arrays

* slicing of multiple dimensional arrays works similarly to lists and strings.
* for each dimension, we can specify a particular slice
* : indicates that everything along a dimension will be used.

## examples

a = np.array([[1, 2, 3], [4, 5, 6]], float)
a[1, :] ## row index 1

## array([4., 5., 6.])

a[:, 2] ## column index 2

## array([3., 6.])

a[-1:, -2:] ## slicing rows and columns

## array([[5., 6.]])

## reshaping

An array can be reshaped using the reshape(t) method, where we specify a tuple t that gives the new dimensions of the array.

a = np.array(range(10), float)
a = a.reshape((5,2))
print(a)

## [[0. 1.]
## [2. 3.]
## [4. 5.]
## [6. 7.]
## [8. 9.]]

## flattening an array

.flatten() converts an array with a given shape to a 1-D array:

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(a)

## [[1 2 3]
## [4 5 6]
## [7 8 9]]

print(a.flatten())

## [1 2 3 4 5 6 7 8 9]

## zero/one arrays

* np.zeros(shape) and np.ones(shape) work for multidimensional arrays if we provide a tuple of length > 1
* use np.ones\_like(), np.zeros\_like(), or the .fill() method to create arrays of just zeros or ones (or some other value) and are the same shape as an existing array

b = np.ones\_like(a)
b.fill(33)

## identity matrices

* Use np.identity() or np.eye() to create an identity matrix (all zeros except for ones down the diagonal)
* np.eye() also lets you fill in *off-diagonal* elements

print(np.identity(4, dtype=float)),

## [[1. 0. 0. 0.]
## [0. 1. 0. 0.]
## [0. 0. 1. 0.]
## [0. 0. 0. 1.]]
## (None,)

print(np.eye(4, k = -1, dtype=int))

## [[0 0 0 0]
## [1 0 0 0]
## [0 1 0 0]
## [0 0 1 0]]

## array mathematics

* for lists (or tuples or strings), the + operation concatenates two objects to create a longer one
* **this works differently for arrays**
* use np.concatenate() to stick two suitably shaped arrays together: to concatenate two arrays of suitable shapes, the

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
b = np.array([[10, 11,12], [13, 14, 15], [16, 17, 18]])
print(np.concatenate((a,b)))

## [[ 1 2 3]
## [ 4 5 6]
## [ 7 8 9]
## [10 11 12]
## [13 14 15]
## [16 17 18]]

## array operators

* When the + operation is used on arrays, it is applied on an element-by-element basis.
* This also applies to most other standard mathematical operations.

print(a+b)

## [[11 13 15]
## [17 19 21]
## [23 25 27]]

print(a\*b)

## [[ 10 22 36]
## [ 52 70 90]
## [112 136 162]]

print(a\*\*b)

## [[ 1 2048 531441]
## [ 67108864 6103515625 470184984576]
## [ 33232930569601 2251799813685248 150094635296999121]]

## adding arrays and scalars

* To add a number, say 1, to every element of an array a, type a + 1
* similarly for other operations, like -, \*, \*\*, /, . . .

print(a + 1)

## [[ 2 3 4]
## [ 5 6 7]
## [ 8 9 10]]

print(a/2)

## [[0.5 1. 1.5]
## [2. 2.5 3. ]
## [3.5 4. 4.5]]

print(a \*\* 3)

## [[ 1 8 27]
## [ 64 125 216]
## [343 512 729]]

## more math functions

* numpy comes with a large library of common functions (sin, cos, log, exp, . . .): these work element-wise
* some functions that can be applied to arrays
	+ for example a.sum() and a.prod() will produce the sum and the product of the items in a:

print(np.sin(a))

## [[ 0.84147098 0.90929743 0.14112001]
## [-0.7568025 -0.95892427 -0.2794155 ]
## [ 0.6569866 0.98935825 0.41211849]]

print(a.sum())

## 45

print(a.prod())

## 362880

print(a.mean())

## 5.0