C200: Take Home Midterm Fall 2021

Due: Saturday, October 16, 11PM

Drs. Brown and Williamson

This is the take home portion of your midterm exam. You are required to work on this by yourself. You can only use Python features that have been covered in lecture or lab up to the review session (October 6th). You must implement the programs on your own–do not *copy* any existing programs (if they, indeed, do exist).

We will not be answering questions on Piazza that will lead to an answer. If the files are not in your repository, please let us know on Piazza as soon as possible.

For grading purposes (as you have seen in the previous assignments), there will be additional tests that we will run. These parameters we pass in will not be edge cases, just a variation in values (that logically fit with the problem).

Please ensure the file is in GitHub and is pushed before the due date and there are *no* syntax errors.

For all problems, we will be giving partial credit based on the correctness of logic. Each problem will be looked over, however syntax errors that prevent the file or function from running will significantly reduce the partial credit. It is better to have a function that gives back incorrect values versus a function that does not let the program run.

You will complete this before 11pm, Saturday, October 16th 2021.

Prob.	Pts	Pts
1.	25	_____
2.	20	_____
	[45]	_____

1. [25] Automatic speech recognition has allowed smart devices (e.g. iPhones, Google Home assistants, and Amazon Echo) to respond to our voice commands. This is partially possible because ASR approaches model many cues of a language. One of those cues is a count of the number of times a certain word transition occurs in a text document. These counts are modeled using *transition matrices*.

   For instance, in the sentence "The cat is in the house", there are 5 unique word transitions (e.g. "The" → "cat", "cat" → "is", "is"→ "in", "in" → "the", "the" → "house"), where each word transition occurs one time. The word transitions involve 5 unique words ("the", "cat", "is", "in," "house"). In a different example, in the text "The cat is outside, but the cat should be in the house", there are 10 unique word transitions ("the" → "cat", "cat" → "is", "is"→"outside", "outside"→ "but", "but" → "the", "cat" → "should", "should" → "be", "be" → "in", "in" → "the", "the" → "house"), where each transition occurs once except for the transition from "the" →"cat", which occurs twice. Likewise, there are 9 unique words in this example ("the", "cat", "is", "outside", "but", "should", "be", "in", "house").

   An example transition matrix for this latter example is shown in the table below. The rows indicate the first word in the transition, while the columns define the second word in the transition. The number at the intersection of each row and columns indicates how many times that word transition occurred in the text. So from the table, we see that the word transition "the" → "cat" occurred 2 times, the word transition "is" → "outside" occurred once, while the word transition "house" → "in" did not occur.

	"the"	"cat"	"is"	"outside"	"but"	"should"	"be"	"in"	"house"
"the"	0	2	0	0	0	0	0	0	1
"cat"	0	0	1	0	0	1	0	0	0
"is"	0	0	0	1	0	0	0	0	0
"outside"	0	0	0	0	1	0	0	0	0
"but"	1	0	0	0	0	0	0	0	0
"should"	0	0	0	0	0	0	1	0	0
"be"	0	0	0	0	0	0	0	1	0
"in"	1	0	0	0	0	0	0	0	0
"house"	0	0	0	0	0	0	0	0	0

Table 1: Transition matrix for the sentence: "The cat is outside, but the cat should be in the house." Note that a word only appears once along the row and column, though it may occur multiple times in the text.

   For this problem, you need to do the following (see code example on the next page): (1) write a function called `unique_words` that takes in a string of text as an argument, and returns a list of the unique words that occur in the string, (2) write a function called `get_transition_matrix` that takes in a string as input, and returns the transition matrix in the form of a nested list.

   You will not be provided with unit test for this, so be sure to come up with your own tests to ensure that your program works. To simplify the problem, assume that periods and commas, are the only punctuation that could be present in the text. However, you need to also assume that the string can contain many sentences. `HINT:` Get it to work with a single sentence first, before moving on to strings with multiple sentences.

   You can use basic built-in string operations (e.g. `split`, `strip`, `replace`, `lower`, `remove`, ...) to solve this problem.

```python
import numpy as np

def unique_words(xstring):
    pass

def get_transition_matrix(xtr):
    pass

if __name__ == "__main__":
    text =  "The cat is in the house. The dog is outside playing with the ...
             kids. Both the dog and the cat need a bath. The kids need to ...
             come in and eat dinner."

    uniwords = unique_words(text)
    print(uniwords)
    print("There are {0} unique words in the text.".format(len(uniwords)))

    print("The Transition Matrix is Below:")
    # np.array() converts the list to a numpy array for display/print purposes.
    print(np.array(get_transition_matrix(text)))
```

Output

```
['the', 'cat', 'is', 'in', 'house', 'dog', 'outside', 'playing', ...
 'with', 'kids', 'both', 'and', 'need', 'a', 'bath', 'to', 'come', ...
 'eat', 'dinner']
There are 19 unique words in the text.

The Transition Matrix is Below:

[[0 2 0 0 1 2 0 0 0 2 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

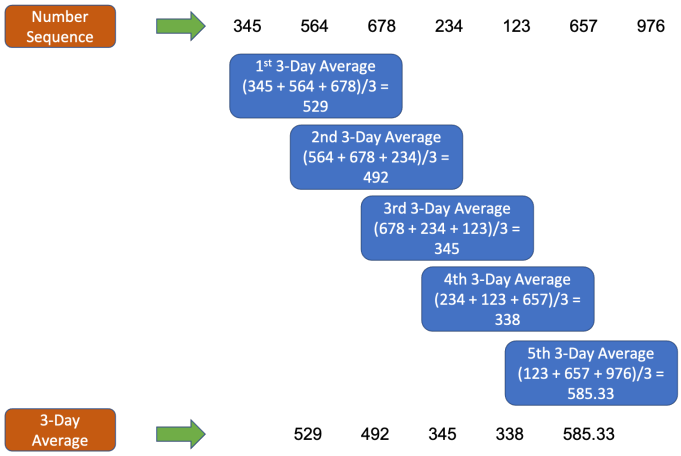

Figure 1: Example showing how the moving 3-day average is computed over a number sequence

.

2. [20] A running or moving average is a way of analyzing how smaller subsets of data evolve over time. Please visit `https://en.wikipedia.org/wiki/Moving_average`. For instance, the number of COVID-19 cases is often analyzed in terms of the 7-day running average. A depiction of computing the 3-day running average for a provided number sequence is shown in the above figure.

   For this problem, you are to write a function called `running_average` that receives an arbitrary length list of numbers and a number indicating the period for computing the running average. Your function should output a list that contains the running average based on that period.

```
1  def running_average(xlist,per):
2      pass
3
4  if __name__ == "__main__":
5
6      #Generate random data sequence
7      data = [11, 82, 91, 55, 32, 91, 12, 5]
8      print(data)
9
10     period = 3 # time period for running avg (3 day average)
11     run_avg = running_average(data,period)
12
13     print("The {0}-day running average is: {1}".format(period,run_avg))
```

Output

```
[11, 82, 91, 55, 32, 91, 12, 5]
The 3-day running average is: [61.33, 76.0, 59.33, 59.33, 45.0, 36.0]
```