Problem Set 4

Due Monday Oct. 14, 10 am

Comments

- This covers material in Unit 5, Sections 7-9.
- It's due at 10 am (Pacific) on Monday October 14, both submitted as a PDF to Gradescope as well as committed to your GitHub repository.
- Please see PS1 for formatting and attribution requirements.

Problems

1. *Memoization* of a function involves storing (caching) the values produced by a given input and then returning the cached result instead of rerunning the function when the same input is provided in the future. It can be a good approach to improve efficiency when a calculation is expensive (and presuming that sometimes the function will be run with inputs on whic it has already been used). It's particularly useful for for recursive calculuations that involve solving a subproblem in order to solve a given (larger) problem. If you've already solved the subproblem you don't need to solve it again. For example if one is working with graphs or networks, one often ends up writing recursive algorithms. (Suppose, e.g., you have a geneology giving relationships of parents and children. If you wanted to find all the descendants of a person, and you had already found all the descendants of one of the person's children, you could make use of that without finding the descendants of the child again.)

   Write a decorator that implements memoization. It should handle functions with either one or two arguments (write one decorator, not two!). You can assume these arguments are simple objects like numbers or strings. As part of your solution, explain whether you need to use `nonlocal` or not in this case.

   Test your code on basic cases, such as applying the log-gamma function to a number and multiplying together two numbers. These may not be realistic cases, because the lookup process could well take more time than the actual calculation. To assess that, time the memoization approach compared to directly doing the calculation. Be careful that you are getting an accurate timing of such quick calculations (see Unit 5 notes)!

2. Let's work more on how Python stores and copies objects, in particular lists.

   a. Experiment with creating lists of real-valued numbers of different lengths. Try to determine how much memory is used for each reference (each element of the list), not counting the

memory used for the actual values. Also consider lists of more complicated objects to see if the behavior is the same or similar.

b. Use the list `.copy` method with a list of numbers and with a list containing more complicated objects (including another list or some dictionary). Compare the behavior in terms of what happens when you modify elements at different levels of nestedness. Relate the behavior to the help information in `help(list.copy)` and to what we know about how the structure of lists.

c. Consider creating these lists.

```
my_int = 1
my_real = 1.0
my_string = 'hat'

xi = [1, my_int, my_int, 2]
yi = [1, my_int, my_int, 2]
xr = [1.0, my_real, my_real, 2.0]
yr = [1.0, my_real, my_real, 2.0]

xc = ['hat', my_string, my_string, 'dog']
yc = ['hat', my_string, my_string, 'dog']
```

What is different about how the elements in these lists are stored?

3. Suppose I want to compute the trace of a matrix, A, where $A = XY$. The trace is $\sum_{i=1}^{n} A_{ii}$. Assume both X and Y are $n \times n$. A naive implementation is `np.sum(np.diag(X@Y))`.

   a. What is the computational complexity of that naive implementation: $O(n)$, $O(n^2)$, or $O(n^3)$? You can just count up the number of multiplications and ignore the additions. Why is that naive implementation inefficient?
   b. Write Python code that (much) more efficiently computes the trace using vectorized/matrix operations on the matrices. You will not be able to use `map` or list comprehension to achieve this speedup. What is the computational complexity of your solution?
   c. Create a plot, as a function of n, to demonstrate the scaling of the original implementation compared to your improved implementation.
   d. (Extra credit) Implement your more efficient version in using Jax (see Section 9 of Unit 5) and compare the timing to the numpy implementation, both with and without using `jax.jit()`.

4. Suppose we have a matrix in which each column is a vector of probabilities that add to one, and we want to generate a sample from the categorical distribution represented by the probabilities in each column. For example, the first column might be (0.9, 0.05, 0.05) and the second column might be (0.1, 0.85, .0.5). When we generate the first sample, it is very likely to be a 1 (a 0 in Python) and the second sample is very likely to be a 2 (a 1 in Python). We could do this using a for loop over the columns of the matrix, and `np.random.choice()`, but that is a lot slower than some other ways we might do it because it is a loop executing in Python over many elements (columns).

```python
import numpy as np
import time

n = 100000
p = 5

# Generate a random matrix and calculate probabilities.
np.random.seed(1)
tmp = np.exp(np.random.randn(p, n))
probs = tmp / tmp.sum(axis=0)

smp = np.zeros(n, dtype=int)

# Generate sample by column.
np.random.seed(1)
start = time.time()
for i in range(n):
    smp[i] = np.random.choice(p, size=1, p=probs[:,i])[0]
print(f"Loop by column time: {round(time.time() - start, 2)} seconds.")
```

```
Loop by column time: 2.88 seconds.
```

a. Consider transposing the matrix and looping over rows. Why might I hypothesize that this could be faster? Is it faster? Why does it make sense that you got the timing result that you got? Does using numpy's `apply` functionality help at all?

b. How can we do it **much** faster (multiple orders of magnitude), exploiting vectorization? (Hint: This might involve some looping or not, but not in the ways described above. Think about how one can use random uniform numbers to generate from a categorical distribution.)