Problem Set 4

Due Wednesday Oct. 15, 10 am

Comments

- This covers material in Unit 5, Sections 7-9.
- It's due at 10 am (Pacific) on Wednesday October 15, both submitted as a PDF to Gradescope as well as committed to your GitHub repository.
- Please see PS1 for formatting requirements.
- I know those of you taking 201A and 201B have midterm exams the week of Oct. 13. We'll mostly finish covering Section 9 on Monday Oct. 6, so you should be in good position to work on this problem set the week before the midterms.

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/often the function will be run with inputs on which it has already been used). It's particularly useful for for recursive calculations 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.

   Try 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. This problem explores how Python stores strings (and more about lists).

a. Let's consider the following lists of strings. Determine what storage is reused (if any) in storing 'abc' in the two lists.

```
a = ['abc', 'xyz', 'def', 'ghi']
b = ['abc']*4
```

b. Next, let's dig into how much memory is used to store the information in a list of strings. Determine (i) how much memory is used to store a simple string (and how does this vary with the length of the string), including any metadata, (ii) how much memory is used for metadata of the list object, and (iii) how much is used for any references to the actual elements of the list (i.e., how the size of the list grows with the number of elements). Experimenting with lists of different lengths and strings of different lengths should allow you to work this out from examples without having to try to find technical documentation for Python's internals.

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 the full speedup that is possible. 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 using Jax (see Section 9 of Unit 5) and compare the timing to the numpy implementation, both with and without using `jax.jit()`.

4. This problem asks you to efficiently compute a somewhat complicated log likelihood, arising from a computation from a student's PhD research. The following is the probability mass function for an overdispersed binomial random variable:

$$P(Y = y; n, p, \phi) = \frac{f(y; n, p, \phi)}{\sum_{k=0}^{n} f(k; n, p, \phi)}$$

$$f(k; n, p, \phi) = \binom{n}{k} \frac{k^k (n-k)^{n-k}}{n^n} \left(\frac{n^n}{k^k (n-k)^{n-k}} \right)^{\phi} p^{k\phi} (1-p)^{(n-k)\phi}$$

   where the denominator of $P(Y = y; n, p, \phi)$ serves as a normalizing constant to ensure this is a valid probability mass function.

   We'll explore how would one efficiently code the computation of the denominator. For our purposes here you can take $n = 10000$, $p = 0.3$ and $\phi = 0.5$ when you need to actually run your code. Recall that $0^0 = 1$.

   a. Write a basic version using map/apply style operations, where you have a function that carries out a single calculation of f for a value of k and then use map/apply to execute it for all the elements of the sum. Make sure to do all calculations on the log scale and only

exponentiate before doing the summation. This avoids the possibility of numerical overflow or underflow that we'll discuss in Unit 8.

b. Now create a vectorized version using numpy arrays. Compare timing to the basic non-vectorized version.

c. Use timing and profiling tools to understand what steps are slow and try to improve your efficiency. Keep an eye out for repeated calculations and calculations/operations that don't need to be done. Compare timing to your initial vectorized version in (b).