Problem Set 4

Due Monday Oct. 9, 10 am

Comments

- This covers material in Unit 5, Sections 7-9.
- It's due at 10 am (Pacific) on October 9, 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. This problem will have you write a decorator that collects (and optionally reports) timing information on how long it takes the decorated function to run. Note that you can just present a unified solution; you don't need to answer each part separately. Please set up the decorator in a module, but demonstrate use the decorator directly in your qmd file.

    a. Write the decorator so that it times the function and prints timing information out for that single execution of the function, as well as returning the result of the function.

    b. Modify the decorator so that when the function is called with a special "back-door" argument, called `_REPORT`, set to `True`, it **returns** a summary of the times for all previous executions of the function rather than invoking the actual computation. Note that your decorator should work regardless of how many arguments the decorated function takes. As an example of the desired behavior, `myfun(_REPORT = True)` rather than `myfun(x,y)` should cause the summary to be returned.

    Hint: You'll need to fool around with passing additional arguments into the wrapper function defined in the decorator function, and the order of how you do that relative to the `*args` and `**kwargs` will matter.

    Note: The use of the underscore in `_REPORT` is intended to avoid any conflicts in the event that the decorated function itself has a regular argument named `REPORT`. We want to set up the naming in situations like this such that our code won't interact badly with code that would be written by users, which usually means having our naming be "special" in some way.

    c. Set things up so that if one sets a global variable called `TIMING` to be `False`, then no timing is done. I.e., by setting `TIMING` to `True` vs. `False`, we can toggle whether the decorator timing functionality is operating at all.

2. This problem explores how Python stores strings.

   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. Consider multiplying an arbitrary $n \times n$ matrix X and a diagonal $n \times n$ matrix D.

   a. How many multiplications are done if you simply matrix multiply D and X?
   b. In principle, how many multiplications need to be done to obtain the result without doing unnecessary calculations?
   c. How can I use `numpy` functions/methods to compute XD efficiently?
   d. How can I use `numpy` functions/methods to compute DX efficiently?

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).