Problem Set 2

Due Thursday Feb. 13, 12:30 pm

Comments

- I haven't fully worked these problems myself, so if you run into any strange issues, please post on Ed as there could be mistakes/oversights on my part.
- Please submit as a PDF to Gradescope.
- Please generate the PDF using Quarto. Feel free to work in a Jupyter notebook and then convert to Quarto before rendering to PDF. Other formats that look professional and are designed for working with code and math notation may also be fine - just check with me via a public post on Ed first.
- Remember to note at the start of your document the names of any other students that you worked with on the problem set (or indicating you didn't work with anyone if that was the case) and then indicate in the text or in code comments any specific ideas or code you borrowed from another student or any online reference (including ChatGPT or the like).
- In general, your solution should not just be code - you should have text describing how you approached the problem and what decisions/conclusions you made, though for simple problems, this can be quite short. Your code should have comments indicating what each function or block of code does, and for any lines of code or code constructs that may be hard to understand, a comment indicating what that code does.
- You do not need to (and should not) show exhaustive output, but in general you should show short examples of what your code does to demonstrate its functionality. The output should be produced as a result of the code chunks being run during the rendering process, not by copy-pasting of output from running the code separately (and definitely not as screenshots).
- I do not recommend writing initial answers using a ChatBot, as I think you are likely to fool yourself in terms of how much you are learning about Julia and programming concepts/skills more generally. But it's up to you to decide how heavily to rely on a ChatBot. And refining your initial answers using a ChatBot seems like a good strategy. Using your own knowledge and information online to check the results of a ChatBot and using a ChatBot to check your own coding can both be important/useful.

Problems

5. Write a function that sums the elements of a dictionary, but only if the values are all integer or floating point scalar numbers, handled by setting the type of the argument (i.e., a Dictionary with types specified for the keys and values). Hint: you will need to use the `<:` syntax. The keys can be any type.

Now add a function with the same name that concatenates strings or character literals as the values.

0. Enhance your Newton function to allow the objective function to take arbitrary additional arguments, passed along from your function into the objective function.

1. Let's robustify our Newton method. Add error trapping using Julia's exception system. This should catch divergence as well as convergence to a local maximum. Add useful messaging with `@info`, `@warn`, `@error`. Also handle the following situations: (1) when the next value has a higher value than the current value (use backtracking) and (2) when the next value is outside the range of the previous values (use bisection).

2. Set up a few tests for your Newton function using Julia's testing framework.

Tip: Here's a gradient function (written in Python) where Newton's method can diverge from some starting points:

def fp(x, theta=1): ## First derivative - we want the root of this. return np.exp(x * theta) / (1 + np.exp(x * theta)) - 0.5

4. Write a function constructor that creates a version of a function that reports how many times it has been called. Ideally this would work regardless of how many (if any) positional or keyword arguments are used by the function. Additionally, try to manipulate `kwargs` so that the wrapper only reports the number of times run when the wrapped function is called as `myfun(_report=true)`.

If you want a real challenge, try to do this using a macro. I was able to do this using some meta programming (code manipulation) tools in Julia but it took some Googling and experimentation and it was helpful that I had some experience with meta programming in R.

3. Consider this overdispersed binomial likelihood calculation. [Show latex for the function] Write using looping and using vectorization and compare speed. Do the calculation inside and outside a function. Assess the time for the JIT compilation. The following is an inefficient vectorized calculation in Python. Convert it to Julia and improve it before comparing to a non-vectorized version.

normConstVecNaive <- function(n, p, phi) { k <- 0:n loglik <- lchoose(n, k) klogk <- k*log(k) klogk[is.nan(klogk)] <- 0 nmklognmk <- (n-k)*log((n-k)) nmklognmk[is.nan(nmklognmk)] <- 0 logLik <- lchoose(n, k) + klogk + nmklognmk - n*log(n) + phi*(n*log(n) - klogk - nmklognmk) + k*phi*log(p) + (n - k)*phi*log(1-p) return(sum(exp(logLik))) } out2 <- normConstVecNaive(n, p, phi)