Problem Set 3

Due Thursday Mar. 6, 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

[UNDER CONSTRUCTION]

Problem 1

Gaussian processes provide a distribution over functions, allowing us to simulate random time series and images, amongst other things. We'll focus on using Gaussian processes in one dimension.

Consider a set of values $x_1, \ldots, x_n \in \mathfrak{R}$. We will simulate $y \sim MVN(\mu 1, \Sigma)$, where μ is (for simplicity) a scalar mean and 1 is a vector of ones. The covariance matrix, Σ is constructed as $\Sigma_{i,j} = f(x_i, x_j)$ for some positive definite covariance function, f. Some examples are the exponential covariance, $f(x_i, x_j) = \sigma^2 \exp(-d_{i,j}/rho)$ for $d_{i,j} = |x_i - x_j|$ and the squared exponential $f(x_i, x_j) = \sigma^2 \exp(-d_{i,j}^2/rho^2)$. Note that in both cases these are simple stationary covariance functions that are functions of the distance between the two points (more complicated covariance functions are possible). To simulate a random function (in the one-d case, you can think of this as a random time series), we compute

$$\mu 1 + Lz$$

where L is the lower-triangular Cholesky decomposition of Σ and z is a vector of n numbers simulated independently from a standard normal.

Problem 1a

- Use structs and functions with type annotation to implement this in Julia. Make sure to continue to cache information effectively for cases where a user wants to simulate multiple time series. We'll discuss linear algebra further in Weeks 5 or 6. Include logging and exceptions as you did in PS2.

An object-oriented implementation is available in Python here. https://stat243.berkeley.edu/fall-2024/units/unit5-programming.html#classes-in-python

Distances.jl has a Euclidean() distance function.

Problem 1b

Take this grid: $x = 0:0.1:1. Use Float64 and Float16 representations to compute Σ and its Cholesky. See what happens with the coarser and finer grids.

x = 0:0.1:1 xFine = 0:0.01:1

dists64 = pairwise(Euclidean(), x); dists16 = Float16.(dists64); SigmaExp = exp.(.- dists64 ./ rho); SigmaExp16 = exp.(.- dists16 ./ rho); SigmaSquExp = exp.(.- dists64 .^2 ./ rho^2); SigmaSquExp16 = exp.(.- dists16 .^2 ./rho^2);

using LinearAlgebra LExp = cholesky(SigmaExp).L LExp16 = cholesky(SigmaExp16).L LSquExp = cholesky(SigmaSquExp).L LSquExp16 = cholesky(SigmaSquExp16).L # fails with 0.1

One (slightly awkward) workaround is to add a small epsilon to the diagonal of Σ for non-numerically positive definite cases.

Plot a random time series with the finer grid and notice the difference in smoothness between the exponential and squared exponential cases.

Problem 1c

- Allow your code to use GPUs if available on the system (but fall back to the CPU if no GPU is available). Benchmark your code on an SCF GPU (or your own if you happen to have one or have access to one), making sure to report which type of GPU you used for your work. In your time comparisons, also compare to using multiple CPU cores via the threaded BLAS.

Problem 1d

- Suppose your users want to generate many time series at once (e.g., for a simulation study). So the output should be a n x m where m is the number of simulations. Use threaded or multi-process parallelization (your choice) to generate the time series in parallel, using the same underlying cholesky decomposition. [investigate parallel RNG and whether they should just use different seed]

Problem 2

Problem 2a

Write a kernel using Julia that computes a kernel density estimate in two dimensions using a normal density with bandwidth $h = \sigma$ for some input dataset with n data points. Your code should compute the KDE on a grid of points, for m points, with each evaluation point handled in a separate thread. This means that each thread will access the full dataset (more on this in part (c).

You'll presumably want a value of h somewhere between, say, 10 km and 300 km.

Apply your code to the data on the location of wildfires in the US (https://www.stat.berkeley.edu/~paciorek/transfer/fire.cs which contains information on the location, year, and size of fires in the US from the federal government's wildfire database (obtained from kaggle here: https://www.kaggle.com/code/behroozsohrabi/us-wildfires-dataset-exploration) and then modified to remove various columns and project to an Albers equal area projection in meters, so that we can directly work with Euclidean distances.

https://www.fs.usda.gov/rds/archive/catalog/RDS-2013-0009.6

paciorek@smeagol:/scratch/users/paciorek/fire_data

Feel free to subset to a portion of the US of interest to you. And feel free to choose a square region and implement your code assuming a regular square grid of evaluation points to keep things simpler. There are 2.3 million fires in the dataset, most of them quite small.

Problem 2b

Now consider using 10-fold cross-validation to find the best value of h from a (small-ish) grid of potential values (possibly equally-spaced on the log scale rather than the original scale), evaluating the log of the kernel density estimate on the test set. First time this by calling your kernel from part (a) 10 times for each value of h.

Next, include the calculation of the density for each value of h in a loop within the kernel. Call your kernel 10 times. Compare the timing when you avoid having to re-access the data for each value of $h.

(Note that one could go further by embedding the cross-validation in the kernel too…)

Problem 2c.

Now consider using the shared memory amongst the threads in a block via `CuStaticSharedArray`. Load the data in blocks of size equal to the number of threads into the shared array in parallel using the threads in the block. Then use your code from earlier to calculate the density for the block of data for each evaluation point in a thread. Finally wrap this by looping over all the data blocks.

Compare the speed here with Problem 2a.

Note that rather than having each thread handle a single evaluation point on the grid, one could have each thread handle a single data point. This would reduce memory access, but there would then need to be a shared reduction operation, which involves *atomic* calls. Ideally one would first do the partial reduction across the data points within each block in shared memory.