Testing statistical software II: there's a theorem somewhere

Ben Klemens

Last time (entry #177), I discussed the value of adding assertions to your code, to check guarantees that the code should follow.

I talked about ad hoc assertions as part of a program, which thus run every time the program runs. This flows into unit testing proper, wherein a given unit of code is accompanied by a separate unit of tests, which is typically run only when something changes. Which approach you take is mostly a question of logistics. You've got options.

This time I'll make some suggestions on testing numeric and statistical results. This is a lot more difficult than typical CompSci functions, where there is rarely a stochastic element, and if it's broken it's blatantly obvious. When you run your Bayesian updating via MCMC search and get a posterior with mean 3.218, is that right? In situations like this, there's no direct way to prove that the functions used to produce that result are correct, but there are always many means of increasing confidence.

Brainstorming some theorems The reason that you are writing code to begin with is that there is a theorem somewhere indicating that the method you are using to calculate a value is somehow useful. That theorem probably says that the result you're getting equals some known and useful value or has some useful property. For example, if you wrote a matrix-inverse function, you know there is a theorem that states that $X \cdot Inv(X) = I$ for any non-degenerate X (but see next time about tolerances).

So let's brainstorm some theorems in some typical situations. Few would be something in a textbook with the heading *Theorem*, but most will be something you can prove or verify relatively easily.

We can start with the special cases: if the input is the identity matrix or zero, it is often trivial to calculate the output. So there's a petit theorem, that the identity matrix or zero will produce a given output, which we can write up as our first test. Such easy and obvious things are a good first place to start, because it is especially easy to set up the inputs and outputs for the test.

For simpler calculations, you may be able to go beyond trivial inputs to a haphazard example you calculated by hand, in which case you have another petit theorem looking something like *if the input is* $[2, 3, 4]$, *then the output is* 12.2.

Can you say something about all possible outputs? Maybe they should all be greater than zero, or otherwise bounded. Maybe the output can't be greater than the sum of the inputs. If the input has some sort of symmetry, will the output be symmetric? With these little theorems, we can write a test that generates a few thousand random inputs and checks that the condition holds for all of them.

John D Cook wrote a book chapter on testing random number generators[1]. The chapter is ba-

[1] `http://www.johndcook.com/blog/2010/12/06/how-to-test-a-random-number-generator-2/`

sically a brainstorming session about things that can be confidently stated about a distribution. For example, there is a theorem that a CDF constructed from draws from a distribution should be 'close' to the original distribution, so a Kolmogorov-Smirnov test comparing the two should give a good test statistic. [He has posted lots of other interesting little tidbits on testing[2].]

Delta Sometimes we know how the output should change given changes in the input. The same Kolmogorov-Smirnov test can be used to express the confidence with which we believe a data set was drawn from a given theoretical distribution. Draw 1,000 values from a Normal$(0, 1)$. I may not know exactly what the p-value will be for a test comparing that to a proper Normal$(0, 1)$, but I do know that that p-value will get larger when looking at a Normal$(.1, 1)$, and it will get larger still when compared to a Normal$(.2, 1)$. If outputs from my K-S test statistic calculator don't follow that pattern, something is wrong.

For that introductory example where the posterior mean is 3.218, you probably have control of the prior mean, and the posterior mean should rise as your prior mean rises (but not as much as you raised the prior mean).

If I run an ordinary linear regression on random data, I'll get some set of results. I may not know what they look like, but I do know that doubling all of the values (X and Y, rescaling the whole problem) shouldn't change any of the slopes, and will double the intercept. This is another test I can run on a thousand random inputs.

Inverses We often have theorems about inverses: given your function f, there is a function g such that we are guaranteed that $g(f(x)) = x$. If such a function exists, then there's another test. For example, if I wrote a matrix-inversion function, I expect that in non-pathological cases Inv(Inv(X))=X (again, see next time about tolerances).

Much of Apophenia[3]'s model testing looks like this. I make a thousand random draws from a model given parameters P, that produces a data set D. I know that if I estimate the parameters of the same model using the data set D, then the calculated parameters should be close to P. I basically define the RNG to be the function such that this (fix parameters–draw–estimate parameters) round trip works. So there's a test.

Now map that same set of a thousand random draws $d_1, d_2, \dots$ to a set of a thousand CDF values, $CDF(d_1), CDF(d_2), \dots$. This will be Uniformly distributed, so there's another test of RNG + CDF.

Especially with these round-trips, I find brainstorming for petit theorems to be pretty fun. Writing code that works reliably requires a lot of time doing mechanical things closer to transforming square pegs so they'll fit in round holes than actual math. A good test takes a function that does some nontrivial calculation, then does another completely unrelated calculation, and produces exactly the expected result. When one of these sets of transformations work for the first time, it feels less like a petit theorem and more like a petit miracle. For a few moments, I feel like the world *works*. I get that buzz that drives us to do more math.

Write everything twice Sometimes, the theorem proves that the efficient and graceful method you just implemented is equivalent to a tedious and computationally intensive alternative. So, there's a test. It's unfortunate, because now you have to sit down and code the ungraceful long way of doing

[2]http://www.johndcook.com/blog/tag/quality/
[3]http://apophenia.info

things. At least it only has to work for a handful of test cases. E.g., for exponential families, you can Bayesian update by looking up the posterior on the conjugate tables, or you can do MCMC, and the output distributions should approach identical.

At my workplace, we have a lot of people who are PhDs in their field but amateurs in the coding world, so they don't have a concept of unit testing. Instead, everything reasonably important is *double-coded*, wherein two independent parties independently write the whole darn program. I tell you this for perspective, so you see that double-coding the tedious version of a specific routine to check specific inputs is not so bad.

We might have another code base somewhere else that calculates the same thing, and can compare our results to theirs. Or, we may have a complete run with a certain data set that we've eyeballed closely and are confident is correct. I'm reluctant to rely on this for testing. There is no theorem that the other guy did the math right. Such compare-the-printout tests work as a regression test (see next time), but if you make legitimate changes, like improving precision or rearranging the output, now you've gotta rewrite the checks. I personally rely on such argument by authority only for eyeballing things and for checking print output routines, and even that much has been a headache for me.

Anything sufficiently well understood can have assertions written about it. I have no idea what sort of mathematical result inspired you to write the code you're writing, but I hope that some of the examples above will help you as you brainstorm petit theorems about your own work. As noted last time, none of these tests prove that your code is Correct, but each shows that you wrote something that meets increasingly tight constraints, and gives users a little more confidence that your code is reliable.