Testing statistical software III: the contract

Ben Klemens

9 May 2013

So far, I've given a brief intro to the mechanics of assertions and tests (entry #177), which you can use to increase your own and others' confidence in your code, and I gave some examples of brainstorming theorems (entry #178) to provide constraints that your function's output has to meet.

The thesis sentence to this part is that the tests are the embodiment of a contract you have with the users. Last time, I gave some suggestions about how to test a matrix inversion function, which the bureaucrat would write as a bullet-pointed contract:

- The output will be such that $X \cdot Inv(X) = I$ to within some tolerance (see below).
- If the input is I, the output will be I.
- If the input is symmetric, the output will be symmetric.

There it is in English; the test is the contract in executable form.

Writing a numeric routine isn't expressionist improvisation: you've got to conceive of what the function does before you write it. And the test routine is the formalization of what the function promises. The typical workflow is to write the tests after you write the function to be tested, but that makes no sense here. Because the contract and test are siamese twins, and you wrote the contract before writing the function, it makes sense to write the test before writing the function as well. The term for writing the test/contract first and then writing a function that conforms to it is *test-driven development*, and it's a sensible work habit that should probably be used more (even by myself).

You are going to sit down and write a routine or three to read in a data set, run a regression, and extract output. Same questions: what's the contract you expect, and how much of it can you express as a test ahead of time? Yes, I know that statistical analysis really *is* expressionist improvisation, and if we knew what we were doing we wouldn't call it science, and exploration is an art upon which we mustn't impose structure. But it's much more fruitful when you explore a data set you can state with confidence was read in correctly, and when an alarm rings if you regress Y on Y. Lewis and Clark kept a log[1] of problems they ran into and surmounted; data explorers can too. The big technological plus we modern explorers have is that we can re-execute our log when the next data set comes in.

Also, once you have the contract/test, the documentation almost writes itself. For the special cases you worked out, show them as examples users can cut/paste/verify; for

[1]`http://lewisandclarkjournals.unl.edu/`

the relatively odd things about symmetry and such, present them as additional useful facts. For some cases you may need more; I suggested a lot of tests for a Bayesian updating routine last time, but the sum of them really aren't enough to fully describe what Bayesian updating is about.

Testing the special cases And don't forget the weird special cases. When I started writing numeric software, I'd sometimes brush off the one-in-a-million edge cases, because, hey, what're the odds that something like that could happen. Which is an easy question to answer: if you send five million independent observations to each be run through a function, a one-in-a-million event has a 99.3% chance of occurring at least once.

Even if you are only sending in a handful of data points to a function that doesn't handle the special cases, the Law of Irony Maximization states that you'll hit that one-in-a-million case anyway.

I hope you find a mathematically clean rule for handling all the edge cases. [Hint: your math platform has a floating-point representation for Not-a-Number (NaN), infinity, and -infinity.] But in some cases, you just have to come up with an arbitrary rule. Then, add clauses to the contract/tests/documentation. I dunno, how about:

- If the input is a zero matrix, I return a square of infinities.
- If the input isn't a square matrix, I return NaN.

To pull a line from a pop song[2] about the endogeneity of social relations, you alone decide what's real. But if you express the rules you wrote in the contract/tests/documentation, then users know exactly what they're getting.

As with legal contracts, the special cases sometimes take many more lines to detail than the simple basic idea. So it goes.

Tolerance Of course, $X \cdot Inv(X)$ is exactly the identity matrix to machine precision only when we are very lucky. All of my tests wind up with a series of assertions making use of this `Diff` macro:

```
#define Diff(L, R, eps) {\
    if(fabs((L)−(R))>(eps)) { \
        printf(#L "=%g is too different from " #R "=%g (arbitrary limit=%g).\n", (double)(L)
            , (double)(R), eps); \
        abort(); \
    }\
}

//sample usage:
Diff(2./3, .6666, 1e−2); //pass
Diff(2./3, .6666, 1e−8); //fail
```

The tolerances to which your tests fit are part of the contract, a bit of documentation for those of us who read tests to see what to expect from these functions. Again, you

[2]https://www.youtube.com/watch?v=WUA9fF8RYDI

alone decide the tolerance level to test to, and then it's there in the contract for the user to work with.

I used to try really hard to get `eps` as small as possible, but Apophenia's tests ship to the public, and I'd get emails from people telling me that on their machine, some test produces a result where `L-R` in the above was 1e-4, but the test listed the max tolerance as 5e-5. It would be nice if that could be improved, but it's very low on my to-do list. I'd typically change all the internal variables from `double` to `long double` and consider that case closed. I leave the tolerances looser now.

Anyway, once you've achieved a reasonable level of accuracy, nobody really cares. In all the endless promotion and bragging about how awesome the latest awesome math package is, how often do you see anybody brag that their package is *more precise* than the others? We can't really do statistics on arbitrary-precision systems (yet), and those people who truly rely on high precision so they can send their probe to Mars need to do the due diligence of retesting the code anyway.

But tolerances are not to be brushed off entirely, because sometimes these little things really do indicate errors. Maybe you should have divided by $n-1$ instead of n.

I may be the only person to have calculated the exact formula for unbiased sample kurtosis[3] (PDF) without recourse to Taylor expansions—it is not a pleasant traipse through mathemagic land. There was a draft of the `apop_vector_kurtosis` function where the best tolerance I could get was around $1e-2$. In this case, the low tolerance really was telling me something: there was a math typo in my transcription of the messy formula. The test cases now pass to within a tolerance around $1e-5$ which is stellar, given that everybody else uses an approximate formula from the start. Tell all your friends: Apophenia has the most precise unbiased sample kurtosis around.

Coverage When I run my main test script, I cover over 90% of the code, which I know thanks to `gcov`, a coverage checker that accompanies the GNU Compiler Collection. It lists which lines get hit and how often, which means I know which lines are still completely untested. Because if every piece of code is covered by a contract/test, what does it mean when we find code that isn't tested?

Once I've got the list of untested lines, I've got decisions to make about whether they should be tested and how (or whether to cut them entirely, because they're not part of a contract). There's a sort of consensus[4] that 85% is about the right amount of coverage, and it's not worth shooting for 100%. And I concur: being endowed with only finite time, writing tests to cover that last 15% is typically low priority. Anyway, given that line count is meaningless[5], percent coverage can also be gamed and is not a statistic worth really obsessing over.

[Many platforms have a coverage checker of some sort, that behave analogously to gcov for C. However, no coverage checker is available for some computing platforms for which there are rather complex packages available. This freaks me out. If the authors of a complex package have no assurance that every part of the package was tested, then you as a user of that package have no such assurance.]

[3] `http://modelingwithdata.org/pdfs/moments.pdf`

[4] `http://stackoverflow.com/questions/194836/what-is-the-code-coverage-percentage-on-your-project/194848`

[5] `http://bureauphile.wordpress.com/2013/10/24/500-million-lines-of-code/`

Regression testing Every time I make a change in Apophenia, I re-compile and re-run every piece of public code I have written that relies on it, including the ships-with-the-library tests, the examples and semi-public solutions for *Modeling with Data*, the githubbed code for this site[6], the examples from the documentation, and a few other sundry items. If the tests pass, I know that my change has not broken any of the contracts to date.

If I had stuck to ad hoc tests and eyeballing things, I'd have nothing. But as the code base grew, the size of the little side-tests I added to the master test script grew at a corresponding rate. Small side projects have a few small side-tests; the magnum opera have a formidable automated list of checks.

Back at the beginning of part one, I recommended some simple checks, like whether all the entries in the age category are positive and under 120. Here at the end, those tests have added up to a contract that not only covers everything in the code base, but that can actually guide how the code is written to begin with. The work is much more challenging here in floating-point land, because the larger units can be very difficult to test, and even small blocks require some creativity in coming up with little theorems to assert about outputs, and we will always be only within some tolerance of the true value anyway. But don't let that daunt you: because it is rarely obvious when a calculated number is incorrect, we should be especially vigilant in adding tests and assertions to raise our confidence.

Next time I'll give a relatively simple example, which is mostly an excuse to post some code that I've been meaning to post for a long time.

[6] `https://github.com/b-k/modeling_examples`