RNG-based models

Ben Klemens

15 July 2013

[Part of a series of posts that started here (entry #146)]

Next time: the big denouement, where we finally get to agent-based models. But first, one more twist on the standard model form.

Say that nobody ever worked out a Central Limit Theorem. We are confident that if we sum a thousand zero-one coin flips, we would get some sort of consistent distribution. Maybe we can say that the expected value of the total will pretty certainly be around 500, but we aren't sure of anything beyond that.

You could write an RNG to simulate the scenario pretty easily (I'll have an example below). A good random number generator makes draws proportional to the likelihood function that it is making draws from, which means that there is a one-to-one mapping between RNGs and likelihoods.

You saw the model objects discussed earlier, and the nice feature that if you define one element, the others could be derived: given a likelihood function, we can write a maximum likelihood search function to do the search; given an RNG or a likelihood, we can calculate the CDF; given the CDF, we can go back to the likelihood.

This entry will be about developing an estimation routine given an RNG. It is a very nontrivial problem, but then, so is the problem of finding a maximum likelihood estimate given a likelihood. For that problem, there are several options (all of which you can tweak in Apophenia by adding an `apop_mle_settings` group to your model):

- You could use gradient descent methods, like conjugate gradients or Newton's method. For a quadratic function, these find the optimum in one (1) step. They are effective to the extent that the function you are optimizing can be approximated by a quadratic function. These methods require a derivative (though if your model doesn't explicitly include one, the `apop_mle` function calculates them via numeric gradients).
- Use the Nelder-Mead Simplex algorithm. This sets a proposed boundary around the optimum and moves one corner of the boundary at every iteration. Sometimes the border walks like an amœba, and when it approaches convergence, it shrinks down toward the optimum. This method requires no derivatives, and doesn't have any smooth concavity conditions. I've found that it can sometimes get stuck, though.
- Use simulated annealing. This method does a sequence of random jumps—it is a variant of the Metropolis-Hastings algorithm. The jumps gets gradually

smaller, so if there are multiple local optima, the search may jump between the hills around each early on in the pattern, but then it will eventually make small enough jumps that it stays on only one hill and climbs its way to the top. It can be shown that, given infinite time, it finds the global optimum with probability one.

- Use an EM-style search. Come up with a starting value for all N dimensions of the parameter vector being searched. Fix dimensions $2, \dots, N$ at the starting values, then do a one-dimensional search of the first dimension, using any of the above methods. Then fix the first dimension at that point, and do a search of the optimum for the second dimension. Repeat, cycling through the dimensions, until convergence.
- Tweak the settings: what's the starting point? What's the tolerance before we accept that convergence has occurred? The initial step size? For simulated annealing, what's the cooling schedule? For an EM-style search, what is the overall tolerance?
- Restart. Start with a search with a broad tolerance and large step size, then restart at the optimum found there with a small step size and small tolerance.

All these options and variants exist because there's no single right method.

Estimation with an RNG If we have only an RNG, we could generate an approximate likelihood rather easily:

- Make 1,000 random draws given the parameters.
- Apply a Kernel density, moving average, cubic spline, or whatever other smoother to turn those thousand spots into a continuous curve.
- Call that your likelihood function.

Conceptually, this isn't difficult at all. But now we have more engineering questions: is 1,000 draws right, or 100, or 10,000? Which smoothing method, if any, should we use?

Now we have a functioning black-box likelihood function, we can use an ML search as above to find the optimal parameters given a black-box likelihood function and a data set.

There are no conceptual problems, only engineering annoyances layered on top of engineering annoyances. I'd like to provide a happy ending here, and assure you that there's an easy formula for selecting all these settings, but if there is one, I haven't found it yet. Here are a few pointers, though.

Srsly, this is expensive Scripting languages are sometimes surprisingly slow—not the couple of percent slower users guess they're going, but slower by a factor of hundreds. Estimation using an RNG is especially expensive: the MLE search tries new parameter vectors (possibly hundreds of them), and for each new parameter vector, we have to generate a new likelihood distribution, which means kernel-smoothing a few thousand new draws. And that's why Apophenia is written in C.

Conjugate gradients won't work Derivative-based methods get very confused by a stochastic likelihood function, and are guaranteed to produce imprecise results. I've found the Simplex algorithm works OK in practice. Simulated annealing is well-suited for a stochastic likelihood (but see the next point), although it will always be the slowest option given a reliable cooling schedule.

Rejection sampling with an RNG Rejection sampling involves making a draw, marking its likelihood as L_1, then making another with likelihood L_2, and jumping to the second point if $L_2 > L_1$, or with probability L_2/L_1 otherwise. Typically, we calculate L_1 once, because it isn't stochastic, so why bother with the redundancy of recalculating it. If L is stochastic, then we could get artificially stuck: say that the set of random draws when first finding L_1 did especially well, inflating the estimate of L_1 a little more than it merits, meaning that more subsequent samples will be rejected than should be. At an extreme case, the random walk may get stuck for a long time on one lucky draw. The solution is to re-calculate L_1 every time it is used as a candidate. This is computationally expensive but necessary. Also, the source code for Apophenia needs to be tweaked to allow this as an option for simulated annealing or Bayesian updating.

A Sample Here's some sample code, generating a Bernoulli distribution using random draws. I put comments into the code itself this time.

```
#include <apop.h>

//Binomial draws. Input parameter gives the odds of a one; we always make 1,000 draws.
void rng(double *out, gsl_rng *r, apop_model *m){
    double prob = apop_data_get(m−>parameters);
    int draws = 1000;
    *out = 0;
    for (int i=0; i< draws; i++) if (gsl_rng_uniform(r) < prob) *out += 1./draws;
}

//For centering a uniform distribution around a point.
//Cut/pasted from the Apophenia documentation, but narrowed range to .0.8. This also took
    some tweaking.
//The uniform is not a good choice (and ruins the covariance estimate), but the premise
//was that we don't know the formula for a Normal distribution.
void set_midpoint(apop_data * in, apop_model *m){
    apop_data_set(m−>parameters, 0, −1, apop_data_get(in)−0.07);
    apop_data_set(m−>parameters, 1, −1, apop_data_get(in)+0.07);
}

//The probability: draw from the rng, smooth with a kernel density, calculate p.
long double p(apop_data *d, apop_model *m){
    int draw_ct = 100;
    apop_data *draws = apop_model_draws(m, draw_ct);
    apop_model *smoothed = apop_model_copy_set(apop_kernel_density, apop_kernel_density
        ,
            .base_data =draws, .kernel=&apop_uniform, .set_fn=set_midpoint);
    double out = apop_p(d, smoothed);
    apop_data_free(draws);
    apop_model_free(smoothed);
    return out;
}

apop_model binom = {"Binomial draws (n=1000) via random draws", .vsize=1, .dsize=1, .
    draw=rng, .p=p};

//Now let's use the model: make five draws from it, find the probability of those
//draws given various paramter values; find the optimal parameter given the input data.
int main(){
    apop_data *five_draws= apop_data_alloc(5,1);
    asprintf(&five_draws−>names−>title, "five draws");
    apop_model_draws(.model=apop_model_set_parameters(binom, 0.3),
                    .rng=apop_rng_alloc(123), .draws=five_draws);
    apop_data_print(five_draws);
    printf("\n\n");

    #define showprob(p) printf("PDF(five draws|param=" #p ") = %g\n", \
            apop_p(five_draws, apop_model_set_parameters(binom, p)));

    showprob(0.2)
    showprob(0.25)
    showprob(0.3)
    showprob(0.35)
    showprob(0.5)

    printf("\n\n");
    Apop_model_add_group(&binom, apop_mle, .step_size=0.1, /*.method=
        APOP_SIMPLEX_NM,*/
       .tolerance=1e−7, /*.verbose='y',*/ .starting_pt=(double[]){0.4});
    apop_model_print(apop_estimate(five_draws, binom), NULL);
```