Uncertain ABM settings

Ben Klemens

24 July 2013

Last time (entry #155), I talked about how a good narrative model accommodates uncertainty all the way through, as opposed to a model that posits a deterministic relationship $f(x)$ and then fuzzes it into $f(x) + \epsilon$, where ϵ is random error.

Bayesian Updating is one means of adding uncertainty earlier in the pipelin, but it has a specific form: we start with a prior model with parameter and data space $\mathbb{P}_p$ and $\mathbb{D}_p$ and a likelihood model with spaces $\mathbb{P}_L$ and $\mathbb{D}_L$. If $\mathbb{D}_p = \mathbb{P}_L$, the it makes sense to draw $RNG_p(p)$, with $p \in \mathbb{P}_p$, and then accept or reject them based on the likelihood $L_L(d, p)$, where $d \in \mathbb{D}_L$. This formulation is a bit like a stream of water with a valve, or a transistor: a stream of draws from $\mathbb{D}_p$ comes through, and the likelihood allows some portion of the flow through but blocks other parts. The final product must be a set of observations in $\mathbb{D}_p = \mathbb{P}_L$, or if you can parameterize it, new parameters in $\mathbb{P}_p$.

But there may be other parts to a model. Here, I make a distinction between *parameters* and *settings*, where I define those two classes of variable with a question: would you optimize over the variable with a maximum likelihood search? If yes, then it's a parameter; else it's a setting. For example, our model might put out a histogram, and the number of bins is worth tweaking but typically not optimized. A noncentral t distribution can be defined to have mean and variance parameters μ and σ, but one typically does not optimize over the degrees of freedom. Agent-based models invariably have settings like the number of agents or whether they travel on diagonals or the number of other agents an agent can connect with in a period. The distinction is a little subjective, and there are cases where one would optimize over the number of bins and such, but that decision defines the model's parameter space, and a different decision defines a different parameter space and thus a different model.

There are people who are skeptical of any model with settings. There's a certain simplicity to a Normal distribution with its two parameters and zero settings—even though it's really a noncentral t distribution with the degrees of freedom (df) setting fixed at the clearly incorrect value of $df = \infty$. It is clear that nobody cheated by picking a convenient value of the settings because there are no settings to set.

This is an æsthetics argument, not really about scientific inquiry. But there is one thing we can do: we can add uncertainty to the settings. It's not a prior, because in today's example the likelihood function (a data-composition between a search microsimulation and a Weibull distribution) has three spaces: $\mathbb{P}_L$, $\mathbb{D}_L$, and a settings space $\mathbb{S}_L$. We'll add a 'prior', with $\mathbb{D}_p = \mathbb{S}_L$, and then observe the distribution of parameters ($\in \mathbb{P}_L$). There is no explicit prior distribution over $\mathbb{P}_L$ but we'll get one as output.

A spatial ABM of search In today's example, we'll have two types of agents. You can think of them as boys and girls, or employees and employers, or just Type A and Type B.

This is a spatial search model:

- An equal number of agents of types A and B are randomly placed on a grid.
- Agents in the middle of the grid have eight neighboring squares; agents on the edge or a corner have fewer because they can't go off the grid.
- Until all agents are paired up:
  - If an agent is adjacent to another agent of the opposite type, they pair up, and are taken out of the model. We record how long it took for them to find each other.
  - Remaining agents take a single step to a random unoccupied neighboring square.
- The model output is the list of pairing times.

We can expect that initially, when there are a lot of agents wandering around, that some agents will find each other quickly. Eventually, we'll be down to only one A agent and one B agent, and they will wander for a long time before pairing up.

In this case, the ratio of agent count to grid size really matters: 90 agents on a 10×10 grid is a very different search from 90 agents on a $1,000 \times 1,000$ grid.

This is the most involved simulation to date, which means that it looks like the most typical C program to date. Also, it's fully 90 lines [though I could play golf and cut the line count some more, at the cost of readability]. Passing on the C trickery (the elaborate macros, the mechanics of `for` loops, and all the pointer use without `malloc` or `free`), here are some notes that might be interesting to an ABMer:

- There is a random number generator associated with every agent. RNGs tend to not be parallelizable, but if every agent has its own, then we can still thread the agents' activities (though I don't do that here).
- The agents live in two structures, set up in `run_sim`. One is a non-changing array of agents. The second is a grid of pointers, with either `NULL` (vacant) or a pointer to one of the agents in the unchanging list of agents. We can easily loop through the agents via the array, and check positions and their surroundings with the grid. When agents are paired up, mark `done=true` in their structs, leave them in the array, and erase their presence on the grid.
- I have exactly as many A agents as B agents, meaning that I don't have to make decisions about what happens with unpaired agents. A more extensive model would need to confront this. This also means that I only have to check for matches and record time-to-exit among one type.
- This is all wrapped up as a first-class model, with $\mathbb{P} = \emptyset$ and $\mathbb{D}$ representing the pairing times of each agent.

At the end of this, I have a list of agent exit times. The rate at which agents exit changes with time. This is also the story of a Weibull distribution, so it'd be interesting

```
#include <apop.h>
#include <stdbool.h>

typedef struct{
    gsl_rng *rng;
    int x, y;
    char type;
    bool done;
} agent_s;

#define gridpt(x, y) grid[(x)*grid_size + (y)]

#define xoff_yoff_loop(...) \
    for (int xoff=-1; xoff <=1; xoff++) \
        for (int yoff=-1; yoff <=1; yoff++) { \
            if (a->x + xoff >= grid_size || a->x + xoff < 0 || \
                a->y + yoff >= grid_size || a->y + yoff < 0 || (!xoff & !yoff) ) \
                continue; \
            __VA_ARGS__ \
        }

agent_s *search_for_mate(agent_s *a, agent_s **grid, int grid_size){
    xoff_yoff_loop (
        agent_s *b = gridpt(a->x+xoff, a->y+yoff);
        if (b && b->type!=a->type) return b;
    )
    return NULL;
}

void step(agent_s *a, agent_s **grid, int grid_size){
    int open_ct = 0;
    xoff_yoff_loop (
        if (!gridpt(a->x+xoff, a->y+yoff)) open_ct++;
    )
    if (!open_ct) return; //landlocked, can't move
    int move = gsl_rng_uniform(a->rng) * open_ct;
    xoff_yoff_loop (
        if (!move--) {
            gridpt(a->x, a->y) = NULL;
            a->x += xoff;
            a->y += yoff;
            gridpt(a->x, a->y) = a;
            return;
        }
    )
}

void generate_agents(agent_s **grid, int grid_size, int pop_size, agent_s *out){
    for(int i=0; i< pop_size; i++){
        agent_s *a = out+i;
        *a = (agent_s){.rng = apop_rng_alloc(apop_opts.rng_seed++),
                       .type = (i % 2) ? 'A' : 'B'};
        do{ //find a vacant spot
            a->x = gsl_rng_uniform(a->rng) * grid_size;
            a->y = gsl_rng_uniform(a->rng) * grid_size;
        } while (gridpt(a->x, a->y));
        gridpt(a->x, a->y) = a;
    }
    out[pop_size] = (agent_s){}; //empty stopper.
}
```

to see how well one fits. A Weibull has two parameters: λ follows your intuition from Exponential or Poisson distributions: if $k = 1$, then λ is the mean time to exit. If $k < 1$, then the time to exit is slower for those that are still present later in the game: the more attractive (or more centrally-located) get picked up quickly, and the hard-to-match take several λs' time to find a match. So for this simulation, we expect that λ grows as the grid gets more sparse, and k should be noticeably less than one.

The code below sets up a Weibull model as a first-class model. As with most models that assume iid, the likelihood of one element can be written easily, and then the total likelihood is the sum of the map of that function onto every element in turn. I had to add a constraint to keep the MLE search routine from setting λ or k to zero.

The `one_run` function sets the settings for the simulation, makes a few thousand draws, and estimates the parameters of a Weibull using those draws. It's just warm-up so we know that everything works.

The real fun is in putting a prior on the grid size and population settings, as the `fuzz` function does. The output is a PMF of 100 Weibull parameters [produced using maximum likelihood estimation based on 1,000 draws from a simulation that typically runs for hundreds or thousands of periods—and it still runs fast enough on my cheap laptop].

Given the output draws of (λ, k), you can plot and admire them, as I did. [I'm sticking to my policy of not displaying output, because it's the last incentive I can give to get you to run the code yourself.]

```
#include <apop.h>
extern apop_model search_sim;

double one_weibull(double d, void *params){
    double lambda = apop_data_get(params, 0);
    double k = apop_data_get(params, 1);
    return logl(k) - logl(lambda)
        + (k-1)*(logl(d) - logl(lambda))
        - powl(d/lambda, k);
}

static long double positive_params(apop_data *data, apop_model *v){
    return apop_linear_constraint(v->parameters->vector);
}

long double weibull_ll(apop_data *d, apop_model *m){
    return apop_map_sum(d, .param = m->parameters, .fn_dp=one_weibull, .part='a');
}

apop_model weibull = {"The Weibull", .vsize=2, .log_likelihood = weibull_ll,
            .constraint=positive_params};

void one_run(int grid_size, int pop_size){
    printf("------- A run with a %i X %i grid and %i agents:\n", grid_size, grid_size,
        pop_size);
    search_sim.dsize = pop_size;
    double params[2];
    search_sim.more = params;
    params[0] = grid_size;
    params[1] = pop_size;
    apop_model *model_out = apop_estimate(apop_model_draws(&search_sim, 1000), weibull
        );
    apop_model_show(model_out);
}

apop_model *fuzz(apop_model sim){
    int draws = 100;
    gsl_rng *r = apop_rng_alloc(1);
    apop_model *prior = apop_model_stack(
                                apop_model_set_parameters(apop_normal, 10, 2),
                                apop_model_set_parameters(apop_normal, 10, 2));
    apop_data *outdata = apop_data_alloc(draws, weibull.vsize);
    apop_prep(NULL, &sim);
    double params[2];
    sim.more = params;
    for (int i=0; i< draws; i++){
        do {
        apop_draw(params, r, prior);
        } while (params[1]*2 > pow(params[0], 2));
        sim.dsize=params[1];
        apop_model *est = apop_estimate(apop_model_draws(&sim, 1000), weibull);
        Apop_matrix_row(outdata->matrix, i, onerow);
        gsl_vector_memcpy(onerow, est->parameters->vector);
        apop_model_free(est);
    }
    return apop_estimate(outdata, apop_pmf);
}

int main(){
    one_run(10, 10);
```