# What goes into a model?

Ben Klemens

29 June 2013

[Part of a series of posts that started here[1]]

Last time, I'd pointed out how many people think that it's blatantly obvious that statistical models should be expressed in code as an object (as opposed to an *ad hoc* aggregation of functions, environments, and whatever else seems necessary at the moment).

So what methods should go into that object?

There's a balancing to be done: too many elements and you have a mess where some parts are inapplicable to some models and every week somebody else thinks of something new that a model should do; too few elements gives up on the object concept.

**Data, parameters, likelihoods** One definition (herein the 'small' definition) is that a model is a parameterized likelihood function. You give me data and a parameter set (like $\mu$ and $\sigma$ for a Normal distribution), and I give you a likelihood. That is, given the parameter space $\mathbb{P}$ and data space $\mathbb{D}$, the model is a function $L : \mathbb{P} \times \mathbb{D} \to \mathbb{R}$.

I called it the 'small' definition because I'd like to widen it. The small definition is a *category*, a collection of sets and mappings between those sets, with only one mapping, $\mathbb{P} \times \mathbb{D} \to \mathbb{R}$. But what about others? You can probably think of a few useful functions associated with a model for mapping $\mathbb{D} \to \mathbb{P}$, like estimating the optimal parameters from data, or from $\mathbb{P} \to \mathbb{D}$, like an expected value or random draw function.

I wound up with a category consisting of:

- a data space ($\mathbb{D}$),
- a parameter space ($\mathbb{P}$),
- estimation ($Est : \mathbb{D} \to \mathbb{P}$),
- likelihood ($L : \mathbb{D} \times \mathbb{P} \to \mathbb{R}$),
- RNG ($RNG : \mathbb{P} \to \mathbb{D}$), and
- CDF (cumulative distribution function, $CDF : \mathbb{P} \times \mathbb{D} \to [0, 1]$).

The fun thing about categories is that if you have mappings between them, then (set of categories, morphisms from category set to category set) is itself a category. More on that next time. But for now, some discussion of why the 'larger' definition is worth having.

[1]`http://modelingwithdata.org/arch/00000146.htm`

**Default methods** I see the statistics literature as a broad framework plus a long list of little results about efficiencies we can gain given certain special models. Some models (like the logit or probit) are described as having some loss function, and then we expect the computer to use it to find the optimal parameters; some models (like OLS) have parameters that can be solved in closed form, and it would be folly to waste time on an explicit optimization search. Custom leans heavily on those models where there is a closed form solution.

So the model should accommodate this, with default methods for any black-box model, and hooks for all the special methods as they come up. The estimation routine, then, is a plain black-box optimizer unless you tell me otherwise.

Here's a little demo wherein we use the model struct as a plain optimizer. This example isn't even a statistical model, but declares an objective function representing $-(x-2)(x+3)$. This 'model' has one parameter; i.e., its parameter set is a $1 \times 1$ matrix. Having declared the model, finding its optimum is the one-liner it should be.

[You'll need to have Apophenia installed to run the demo code; see Apophenia's setup page[2] for details.]

```
#include <apop.h>

double objective(apop_data *ignored, apop_model *m){
    double param = apop_data_get(m->parameters);
    return -(param-2)*(param+3);
}

apop_model quadratic = {"y = -(x-2)(x+3)", .p = objective, .m1base=1, .m2base=1};

int main(){
    apop_model *estimated = apop_estimate(NULL, quadratic);
    printf("Optimum found at %g\n", apop_data_get(estimated->parameters));
}
```

Because the model has only a probability method defined, the estimate function knows to call the optimizing routines.

For many well-known models, it's folly to use an optimizer. The maximum likelihood estimator for the $\mu$ parameter of a Normal distribution is the mean of the input data, so it would be folly to call an optimizer, and the next example explicitly specifies the `estimate` routine. This model has parameters in a vector of size 2. The key point to these two demo snippets is that the object structure already works: the `apop_estimate` call doesn't really change even though one model had an explicit `estimate` routine and the other didn't.

```
#include <apop.h>

apop_model *est(apop_data *d, apop_model *m){
    int n = d->vector->size;
    apop_data_set(m->parameters, 0, -1, apop_mean(d->vector));
    apop_data_set(m->parameters, 1, -1, sqrt(apop_var(d->vector))*(n-1./n);
    return m;
}
```

[2] `http://apophenia.info/setup.html`

```
apop_model normal = {"Estimation only for a Normal distribution", .estimate = est, .vbase
    =2};

int main(){
    apop_data *d = apop_data_fill(apop_data_alloc(6), 3, 10, 12, 8, 9, 1.2);
    apop_model *estimated = apop_estimate(d, normal);
    printf("Parameters are:\n");
    apop_data_print(estimated->parameters);
}
```

So that's the basic thinking here: we should have a slot where we can put a known method from the literature whenever possible, but if we don't have a method, there should be a default to put in. The same goes in other directions too: given a log likelihood, I should be able to produce an RNG; given an RNG, I should be able to produce a CDF; et cetera.

By the way, verifying numeric estimation routines is a real challenge. You can check corner cases that can be calculated by hand, but knowing that my OLS routine works for a set of points that are neatly arranged so that I can do the math by hand is not reassuring. However, if the results of a hand-written method using nontrivial data match the results from a default routine, then that's a much stronger indication that both are correct. You're using the same default method over and over, so be sure to do this on a dozen or so models so that you can raise confidence in the default itself.

**IRL** The rest of this entry is about how Apophenia implements the above, and is not necessarily of interest to everybody. The simple category above would be a `struct`:

```
struct apop_model{
    apop_data *parameters;
    apop_data *data;
    apop_model * (*estimate)(apop_data * data, apop_model *params);
    double (*log_likelihood)(apop_data *d, apop_model *params);
    double (*cdf)(apop_data *d, apop_model *params);
    void (*draw)(double *out, gsl_rng* r, apop_model *params);
} apop_model;
```

The first thing to note is that the estimate routine doesn't have a signature $\mathbb{D} \to \mathbb{P}$, but $\mathbb{D}\times\mathbb{M} \to \mathbb{M}$, because it takes in a model with no parameters (`inmodel.parameters==NULL`) and returns a copy of that model with parameters set.

Due to reality, the `apop_model` structure has many more elements, which break into three categories:

- Functions that are boilerplate for objects: print, prep (more-or-less what gets called for a new model), and a single `char` for error codes. [I think every object should have an `error` element, so you can ask the object itself if it ran into glitches in processing.]
- Things that are technically necessary for a coherent model, like a slot for the size of the parameters (so an MLE knows how big a space to search, for example) and a list of model-specific settings.

- Things that, if I knew then what's so obvious now, would be external functions, including the score (dlog liklelihood) and a prediction function.

That last part deserves further note, which I will get to in a few entries if anybody requests it. There are all sorts of things that need to behave differently for certain models, including Bayesian updating, entropy calculations, or expected value. I wound up making this happen via a virtual table method (yes, this reimplements a C++ feature, but in a manner more appropriate to the situation). You call the `apop_update` function with a beta model and a binomial model; it looks in its lookup table and sees that the beta-binomial has a closed-form posterior; it calls the associated function that makes that happen. Taking this to the extreme, one could implement a model as little more than a label, and then every function above (CDF, RNG, estimate, update, ...) all have hooks that allow different behaviors depending on the label sent in. You could write functions that do a similar sort of per-model dispatch for entropy, expected value, Kullback-Leibler divergence, and who knows what else.

Above, I presented one point where one could put the break between between things that should be taken as inherently part of the model and those that are functions that behave differently given different model inputs. There are a continuum of others, inlcuding small definitions that basically move every special method outside the object to a lookup table.