A sample transformation: truncation, 6 June 2013

Ben Klemens

In the the last entry (entry #148), I enthused about how much mileage we can get with a standard model object and a set of standard transformations thereof—and I haven't even gotten to Bayesian updating and hierarchical modeling, which will come next time.

In this entry, I'll present a demo implementation of a truncate-at-zero transformation. It gives me another excuse to show how more offbeat models can be given entirely standard treatment. For those of you who might want to implement something like this, via whatever platform, you might gain benefit from seeing the method I use to implement transformations. Some of you might find this entry to be TMI, in which case, come back in three days.

A brief apology I'm using Apophenia for all of the examples because, as noted in prior posts, I really don't think there's anything out there like it. C custom is to put a library-specific prefix at the head of every function or structure in a library, to prevent name conflicts, so you'll see a lot of `apop_`s in the code below. It gets monotonous, but is very clear. If you think I'm being a narcissist for writing around the library of stats functions I wrote, I'll hook you up with the people who think I need to do exponentially more self-promotion and you can debate amongst yourselves. At least I didn't name the package after myself [I'm lookin' at you, Ross and Robert (smiley face)].

If you hate C, I encourage you to try reading through the sample code anyway and see if you get the gist even if the extra stars and ampersands seem like line noise (and then get my C textbook so it'll all become clear...).

As a final apology, writing this revealed a bug in the `apop_beta` model, so you'll want to download a recent copy of the library to run this against.

An example: the round trip I'll first walk you through the use of the transformation, and then its construction.

[To run these, you'll need to compile them together. In my series of tips on scientific computing, maybe you recall that tip #1 was to (entry #050, use a makefile). Here's the makefile I use for this pair of files:]

```
CFLAGS =-g -Wall -O3 `pkg-config --cflags apophenia` -DTesting
LDLIBS=`pkg-config --libs apophenia`
CC=c99

149-roundtrip: 149-truncate.o
```

One of my favorite tests of a model is a round-trip where we start with a set of parameters, draw a few thousand elements using the RNG, and then use those elements

as a data set to estimate parameters. If the parameters at the end of the round trip are reasonably close to the parameters we started with, then we have more confidence that the model is at least internally consistent.

Here's a summary and reduction of what happens to the Normal distribution in the sample:

```
apop_model *m = apop_model_set_parameters(apop_normal, 1, 1);
apop_model *mt = truncate(m);
apop_data *draws = apop_model_draws(mt, 2e5);
apop_mode *mt_est = apop_estimate(draws, *mt);
apop_model_show(mt_est);
```

Every line is either a transformation of a model or a use of a model:

- Start with an unparameterized stock `apop_normal`.
- Set its parameters to (1¡—,—¿ 1)¡—,—¿ creating a parameterized model.
- Apply the truncation function to create a truncated¡—,—¿ parameterized model.
- Use the model to make 20,000 draws.
- Use the data and the truncated model (the parameters will be thrown away) to estimate the parameters given the data set. This produces a new model with non-`NULL` parameters.
- Print the estimated model parameters to the screen¡—,—¿ so we can see that the results are reasonably close to the original parameters of (1¡—,—¿ 1).

Here's the actual program. I broke this up into `main` and `round_trip` functions, so I can do the draw-and-estimate routine for all four models.

The transformation As is custom with many C files, it's best to read from the bottom up, as the last function in the file is where execution starts, and that function calls elements declared earlier in the file, which call elements still earlier in the file.

At the bottom, you'll find the transformation function, `truncate_model`. It outputs a model object, so everything above that point describes the model object to be returned. These were the only two items also defined in the file above that makes use of the transformation, so you can see that the rest of this file can remain private, internal workings to the transformation.

The function itself only has to make a copy of the `truncated_model`, store a copy of the original model in the new model (the `apop_model` object has a `->more` pointer just for purposes like these) and wire the two parameter sets together.

The truncated model that gets returned is declared in the line above the `truncate_model` function: it has log likelihood, draw, and prep functions, which are as described. The `prep` function is Apophenia-specific, and is what gets called by the `apop_estimate` function to make sure the parameter set isn't `NULL` and other such housekeeping.

The RNG for a truncated model is straightforward: just keep drawing from the base model until we get a draw that isn't below the cutoff. This may be inefficient, but it's one line of code and works for any arbitrary model. If you have a more efficient method for a specific model, you can substitute it in post-transformation:

```
#include <apop.h>

apop_model truncated_model; //these two will be defined in the next file.
apop_model *truncate_model(apop_model *in, double cutoff_in);

void round_trip(apop_model *m){
    apop_data *draws = apop_model_draws(m, 2e5);
    //Uncomment these two lines if you want to look at the data and have Gnuplot installed:
    //Apop_col(draws, 0, v);
    //apop_plot_histogram(v, .bin_count=100, .output_pipe=popen("gnuplot --persist", "w
        "));

    //this copying and NULLifying is unnecessary; it's just so you know I'm not cheating.
    apop_model *clean_copy = apop_model_copy(*m);
    clean_copy->parameters = NULL;

    apop_model_show(apop_estimate(draws, *clean_copy));
}

int main(){
    printf(" N(1, 1)\n");
    apop_model *m = apop_model_set_parameters(apop_normal, 1, 1);
    round_trip(m);

    printf("\n truncated N(1, 1)\n");
    apop_model *mt = truncate_model(m, 0);
    round_trip(mt);

    printf("\n Beta(.7, 1.7)\n");
    m = apop_model_set_parameters(apop_beta, .7, 1.7);
    round_trip(m);

    printf("\n Truncated Beta(.7, 1.7)\n");
    mt = truncate_model(m, .2);
    round_trip(mt);
}
```

```
#include <apop.h>

double cutoff; //A global variable, so you know this isn't for production use.

double under_cutoff(double in){ return (in < cutoff); }

long double like(apop_data *d, apop_model *m){
    double any_under_cutoff = apop_map_sum(d, .fn_d=under_cutoff, .part='a');
    if (any_under_cutoff) return -INFINITY;

    //apop_cdf wants an apop_data set; we have a bare double
    gsl_vector_view gv = gsl_vector_view_array(&cutoff, 1);

    return apop_log_likelihood(d, m->more)
        - (d->matrix ? d->matrix->size1 : d->vector->size) * log(1 - apop_cdf(&(
            apop_data){.vector=&gv.vector}, m->more));
}

void r(double *out, gsl_rng *r, apop_model *m){
    do apop_draw(out, r, m->more); while (*out < cutoff);
}

void prep(apop_data *d, apop_model *m){
    apop_model *base_model = m->more;
    apop_prep(d, base_model);
    m->vsize = base_model->vsize;
    m->msize1 = base_model->msize1;
    m->msize2 = base_model->msize2;
    m->parameters = base_model->parameters;
}

apop_model truncated_model = {"A truncated univariate model", .log_likelihood= like, .draw=
     r, .prep=prep};

apop_model *truncate_model(apop_model *in, double cutoff_in){
    cutoff = cutoff_in;
    apop_model *out = apop_model_copy(truncated_model);
    out->more = in;
    out->dsize= in->dsize;
    out->constraint= in->constraint;
    return out;
}
```

apop_model *truncated = truncate_model(apop_normal, 0);
truncated->draw = your_super_efficient_normal_draw_function;

Cutting off a chunk of the distribution would change the total density unless we compensate, so if the original probability given parameters p and data point d had been $P_{mod}(d, p)$, then after we cut off everything below cutoff C, the new probability is

$$P_{trunc}(d, p) = \frac{P_{mod}(d, p)}{1 - CDF(C, p)}.$$

Of course, if a data point is below the cutoff, that has probability zero and thus log probability $-\infty$.

So there's the tour of the code: the transformation function outputs a model that wraps the code in a simple model object that does the appropriate transformations to the base model that it holds in its internal storage, then the `main` routine can live up to the promise of egalitarian treatment of the pre- and post-transformation models.

As a final point of foreshadowing of future entries, those of you who got this compiled and running will notice how each run produced a covariance matrix for the parameters.