Making a data structure comfortable

Ben Klemens

9 September 2013

This is the last entry in a series of posts on the design and use of the `apop_data` struct, beginning with this entry (entry #162). Next time, I'll get to some more statistical application, regarding interrater agreement.

The purpose of this five-entry series was to consider all the little details of what makes a good structure for data, and this final episode covers four additions to the structure so far to make this structure extensible and, well, comfortable.

Names I haven't mentioned column and row names, though they are absolutely essential for us as humans to understand our data, and are a key difference between a matrix manipulation library and a data analaysis library. But they largely take care of themselves. When you query from the database, the names get put in place, and then it's my problem as the author of functions like `apop_data_transpose`, `apop_data_stack` and `apop_data_split` to make sure that the names get put in the right place.

The most basic use of names is to use them interchangeably with the numeric index of the row or column:

```
apop_data *d = apop_data_calloc(3, 3); //allocate 3x3 matrix filled with zeros.
apop_data_add_names(d, 'c', "left", "middle", "right");
apop_data_add_names(d, 'r', "top", "middle", "bottom");

apop_data_set(d, 2, 2, 2.2);
apop_data_set(d, .rowname="middle", .colname="right", .val=1.2);
apop_data_set(d, .rowname="top", .col=0, .val=0.0);

//view columns or rows by name:
Apop_data_col_t(d, "right", rightcol);
double rightsum = apop_vector_sum(rightcol);
```

The names are in their own mitochondria-like struct, the `apop_name`, which is rarely if ever seen outside of `apop_data` structs, but provides its host `apop_data` struct with much of its power. As with mitochondria, names being in a separate structure was an evolutionary anomaly, but there isn't much reason to merge them fully into the host structure either.

Also, having a separate struct makes it easier to implement hashes for the text of names, to speed up name lookups. (This is another to-do item for anybody interested in working on Apophenia).

Weights Real-world data sets are often weighted, so our data set will need a weights vector. At my workplace, we have surveys where there are more weights columns than data columns, but those can always be reduced to a single weight column for any given analysis.

So let's add that as another vector to the `apop_data` set.

There used to be separate `apop_ols` and `apop_wls` (weighted least squares) models. But there was no need: if you give me weights, the `apop_ols` functions use them to implement WLS, else it runs as ordinary.

An empirical distribution is one where the probability of an event equals its frequency in observed data. So we take a data set, add weights, and call it a model. The most common weighting is where every element has equal weight, so that is what is assumed if `your_data->weights==NULL`. In either case, `apop_estimate(your_data, apop_pmf)` outputs a PMF model that appropriately wraps the data. See also `apop_data_pmf_compress` which reduces redundant observations to one weighted observation.

Pages At this point, we have rectangular data covered. Apophenia tries to not place restrictions requiring that the vector and matrix have the same number of rows, so you could even have one column of numbers that doesn't quite fit the rectangle.

But some data doesn't fit a single grid at all. Maybe your aggregate model requires some data at the individual level and distinct additional data at the aggregate level. Maybe you have a parameter set that is a vector, a matrix, and a handful of loose scalars.

So the `apop_data` struct includes a pointer to `apop_data`. Suddenly it's a linked list of data sets. Given two rectangles of data as per the examples above, link one to the next (via `apop_data_add_page`) and there you have it: one data set with two distinct pages that you can send to a custom model written around this data configuration.

The MLE routine will gladly optimize over paramters that span several pages. I can't believe I haven't demo'ed this yet, but this is yet another opportunity to build bigger models by taping together simple models.

Also, there's footnotes. Maybe you have the covariance of the main data set, or a list of factor conversions, or even text footnotes referenced in the main data (see *NaN boxing* at this earlier blog post (entry #132) or in 21st Century C[1]). You could put that in a separate `apop_data` set, but everywhere you send the main data, you have to send the footnote table; might as well tack it on to the main data set.

I wound up with this rule: if the name of a data set is in `<html-style angle brackets>`, then it's not data *per se* but information about data. Those operations that act on all pages of a data set (like `apop_data_pack` and `apop_data_unpack`, to de-structure a data set into one long vector) will typically ignore info pages unless you ask them not to. For example, Apophenia's MLE routine has to use `apop_data_pack` to de-structure the parameters, because the GSL's optimization routines work on `gsl_vectors`, so info pages in a parameter set are ignored.

[1] `http://www.jdoqocy.com/click-7085528-11290546?url=http%3A%2F%2Fshop.oreilly.com%2Fproduct%2F0636920025108.do%3Fcmp%3Daf-code-books-video-product_cj_0636920025108_%25zp&cjsku=0636920025108`

The error marker Oh, and one last thing your math textbook didn't tell you about: processing errors happen. The `error` eleemnt is a single character, and if it isn't nul, then something went wrong with processing your data set. This makes for a nice processing flow, because you can directly ask the data set returned by a function whether the process went OK.

Now have some fun So there's the `apop_data` struct. It started with just a rectangular matrix, but expanded to be a vector-matrix hybrid, with a text grid, and weights, and names, and extra pages for additional rectangles of data, and a marker for processing errors. This isn't Nobel-worthy innovation, but designing a structure with enough hooks so that a diverse range of data sets can be handled comfortably and reliably is not entirely trivial, either.

Working with it feels like manipulating building blocks, and it's structured enough that it's easy (albeit sometimes tedious) to write functions manipulating those blocks in predictable ways. I've found working with it to be largely natural, and very predictable. There are many situations where I'm not doing statistics *per se*, but use Apophenia to keep tables of data organized.

Scroll down to the 49 functions currently listed in the `apop_data_`... section of Apophenia's function index[2] to see the sort of utility functions that I've written around the structure. Besides the ones that shunt data around, there are also a few, like `apop_data_summarize`, `apop_data_correlation`, or `apop_data_covariance` that do basic statistical work.

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