The data set: views of a C structure

Ben Klemens

13 August

The Apophenia library is built on two structures, one representing data and one representing models. I've talked a lot about the `apop_model` struct, because it's novel and facilitates novel types of analysis. Conversely, the `apop_data` struct represents numeric and text observations, which is not especially innovative. Nonetheless, most of what we do is simple data manipulation, not modeling *per se*, so having a good structure for data matters.

The next several entries offer notes and how-to on the `apop_data` structure. I will assume that you're interested in the questions about what roadblocks come up in building a structure for data, and may or may not be interested in using Apophenia itself.

Here's the definition of the struct. It has a lot of parts. I'll discuss the rationale and use of all of them over the next several entries.

```
typedef struct apop_data{
    gsl_vector *vector;
    gsl_matrix *matrix;
    apop_name *names;
    char ***text;
    size_t textsize[2];
    gsl_vector *weights;
    struct apop_data *more;
    char error;
} apop_data;
```

In this episode: the matrix.

Structuring your data Your data is held in memory somewhere, in some format. You would write it down on paper as a table, with observations along the rows and each column representing some variable (maybe observation ID, height, weight, gender, race...).

A computer doesn't have the luxury of a two-dimensional page. Data addresses are linear, so the machine will have to represent your 2-D structure in a linear form. From this simple problem statement comes a host of little details.

For example, we might take a 3×4 grid and turn it into the sequence
(0,0) (0, 1) (0, 2) (0, 3) (1,0) (1, 1) (1, 2) (1, 3) (2,0) (2, 1) (2, 2) (2, 3)

Already, I've made a few choices. I'm used to C, so I've used offset numbering, where the index represents how far you are from the first element (or row or column). I'm using *row-major* ordering, meaning that the primary clumps are rows, and any one column of data is split up. Row-major is the C custom, but I find myself applying functions like the log likelihood on a per-observation basis more often than I'm operating on a column in isolation. Not that it really matters much.

A set of elements is *regularly spaced* if the distance between item i and item $i + 1$ is always fixed. Thus, the distance from element (0, 1) to the next in the row, (0, 2), is the same distance between any other row-neighbors, such as (1, 2) and (1, 3). The columns are also regularly spaced: the distance from (0, 1) to (1, 1) is four steps—the size of one row—and this is the same for any pair of column-neighbors. You'll find authors who refer to this jump as the *stride* of the matrix.

A fixed-width jump is among the fastest operations a computer can do. Your computer hardware is optimized to deal with it: after you fetch addresses 600, 602, and 604, a modern processor will pick up the pattern and preemptively fetch what's at 606 before your program even asks for it. In typical cases, it doesn't matter whether the distance is one unit as when traversing a row or a larger stride as when traversing a column.

Rare is the data processing algorithm that doesn't have to fetch every element of a matrix, so these little speedups on fetching one record scale in proportion to the size of the data set.

Other systems do it differently, throwing out some form of regular spacing in exchange for some other benefit. R data frames bundle by columns. Each column may have a different type, but a single column is (typically) a regularly-spaced vector of data. SAS and databases bundle by observations (rows). Each observation is a sequential blob of data of different types. In C-type languages, we could implement this via a struct representing one observation and then building an array or tree of such structs.

The GNU Scientific Library implements matrices in the form I described above, with a data segment that is a simple row-major list of elements and some metadata indicating the row count, column count, starting point, and stride.

Views Say that we have a matrix with this metadata:

- data= a pointer to a list of elements somewhere in memory.
- starting point= (0, 0) in the data set
- rows = 3
- cols = 4
- stride= 4

If you need to find element (2, 3), then this is enough information that we can start at (0, 0), then step forward 2*stride + 3 units. If you try to step more than three rows or four columns, then the system knows to throw an error instead of overreaching. So this metadata is sufficient to turn a pointer to a sequence of elements into a real matrix.

Now we want the column vector that is column 1 in the data. You could build that struct by changing only the metadata:

- data= the same pointer to a list of elements.
- starting point= (0, 1)
- rows = 3
- cols = 1
- stride= 4

Notice how the stride is the size of a row in the original data set, even though a 'row' in a column vector is one element wide. But the math is the same: the element at position (2, 0) in this column vector is at position (starting point) + 2*stride + 0.

I leave as a simple exercise to the reader how to build any other regular shape, like a two-by-two square starting at (1,1), or only row 3.

So you get views of columns, rows, or other regular shapes by simply annotating the data accordingly. What you don't get are irregularly-spaced elements. If you want columns 1, 2, 4, and 8, then you'll need more metadata than we have here; the typical solution is at this point to give up on the elegance of rewriting the metadata for the same raw data and start copying columns into a new structure. If you've been reading along to this point, you can see how this has shaped the recommendation that users build data sets in SQL and then write them once in the correct form to a data set, because options for subsetting after writing to a regular matrix aren't as nice (but Apophenia does provide several, all of which involve data copying).

By the way, it looks like somebody posted some slides about how this is also how NumPy arrays work[1].

Examples This is all about vectors and matrices from the GNU Scientific Library, but I'm going to use Apophenia, for exactly the reason I wrote Apophenia: it provides a nicer UI for the GSL.

For example, the raw GSL provides several functions to slice a GSL matrix[2], to pull a column or row[3], or subset a GSL vector[4]. Apophenia has some convenient wrapper macros for many of these functions. E.g., to get the covariance of columns 2 and 3:

```
gsl_matrix *m = [read in data here];
Apop_matrix_col(m, 2, v1);
Apop_matrix_col(m, 3, v2);
apop_vector_cov(v1, v2);
```

After being declared within the macro, `v1` and `v2` correctly behave as vectors. They are pointers to structs on the stack, so they disappear at the end of scope, and they are metadata re-wrappings of the same data as the original, so changes in the vector view affect the base data.

Here's a full example. This time, the key interesting output will be a principal component analysis of the Amash vote data which first appeared several episodes ago (entry #158). Roughly, a PCA reveals in which column the greatest variation in the data lies. If there's a strong correlation between two columns, then that variation may be along a linear combination of those columns. The example uses `Apop_col_v` (like `Apop_matrix_col` but takes in an `Apop_data` set) to get only the first eigenvector from the matrix of eigenvectors.

The query describes what the three columns of data are. I set Aye=0 for consistency with prior analyses; vote=1 indicates a pro-NSA wiretap leaning. I took the log of contributions, for reasons that will become evident below. The Logit analysis from a few episodes ago is also very different when we replace `contribs` with `log(contribs+10)`. Instead of adding 10 to dodge log(0) errors, you could also insert a `where contribs > 0` clause to the query.

The program runs `apop_data_correlation`, then `apop_matrix_pca`, which puts the eigenvalues in the output data set's vector, and the eigenvectors in the columns of the output data set's matrix.

[1]http://axialcorps.com/2013/08/08/numpy-is-core-lessons-from-python-for-data-science/

[2]https://www.gnu.org/software/gsl/manual/html_node/Matrix-views.html

[3]https://www.gnu.org/software/gsl/manual/html_node/Creating-row-and-column-views.html#Creating-row-and-column-views

[4]https://www.gnu.org/software/gsl/manual/html_node/Vector-views.html#Vector-views

```
#include <apop.h>

int main(){
    apop_text_to_db("amash_vote_analysis.csv", .tabname="amash");
    apop_data *d = apop_query_to_data("select "
            "case when vote='Aye' then 0 else 1 end as vote, "
            "ideology, log(contribs+10) from amash");

    apop_data_show(apop_data_correlation(d));

    apop_data *eigens = apop_matrix_pca(d->matrix);

    printf("\nEigenvalues:\n");
    apop_vector_print(eigens->vector);

    printf("\nPrincipal eigenvector:\n");
    Apop_col_v(eigens, 0, principal);
    apop_vector_print(principal);
}
```

Even with the `where contribs>0` clause, the first eigenvector is the lion's share of the variation in the data, and that eigenvector consists mostly of the `log(contribs)` column. As you can imagine, without logs contributions represent ∼100.00% of the variation in the data.

If you run this, you'll see the immense benefit of fast addressing: even on slow hardware, this runs in a fraction of a second, whereas calculating the PCA by hand can take literally hours.