IX. Doing the Arithmetic Redux with Guttman Patterns

Many posts ago, I asserted that doing the arithmetic to get estimates of item difficulties for dichotomous items is almost trivial. You don't need to know anything about second derivatives, Newton's method iterations, or convergence criterion. You do need to:

1. Create an L *x* L matrix $R = [r_{ij}]$, where L is the number of items.
2. For each person, add a *1* to r_{ij} if item j is correct and i is incorrect; zero otherwise.
3. Create an L *x* L matrix $Y = [y_{ij}]$ of log odds; i.e., $y_{ij} = log(r_{ij} / r_{ji})$
4. Calculate the row averages; $d_i = \Sigma y_{ij} / L$.

Done; the row average for row i is the logit difficulty of item i.

That's the idea but it's a little too simplistic. Mathematically, step three won't work if either r_{ij} or r_{ji} is zero; in one case, you can't do the division and in the other, you can't take the log. In the real world, this means everyone has to take the same set of items and every item has to be a winner and a loser in every pair. For reasonably large fixed form assessments, neither of these is an issue.

Expressing step 4 in matrix speak, $Ad = S$, where A is an LxL diagonal matrix with L on the diagonal, d is the $Lx1$ vector of logit difficulties that we are after, and S is the $Lx1$ vector of row sums. Or $d = A^{-1}S$, which is nothing more than the d are the row averages.

R-code that probably works, assuming *L*, *x*, and *data* have been properly defined, and almost line for line what we just said:

Block 1: Estimating Difficulties from a Complete Matrix of Counts R

```
R = matrix (0, L, L)                  # Define and zero an LxL matrix
for ( x in data)                      # Loop through people
    R = R + ((1–x) %o% x)             # Outer product of vectors creates square
Y = log (t(R) / R)                    # Log Odds (ji) over (ij)
d = rowMeans(Y)                       # Find the row averages
```

This probably requires some explanation. The object *data* contains the scored data with one row for each person. The vector *x* contains the zero-one scored response string for a person. The outer product, %o%, of *x* with its complement creates a square matrix with a $r_{ij} = 1$ when both x_j and $(1–x_i)$ are one; *zero* otherwise. The log odds line we used here to define *Y* will always generate some errors as written because the diagonal of *R* will always be zero. It should have an error trap in it like: *Y = ifelse ((t(R)*R), log (t(R) / R), 0).*

But if the *R* and *Y* aren't full, we will need the coefficient matrix *A*. We could start with a diagonal with *L* on the diagonal. Wherever we find a zero off-diagonal entry in *Y*, subtract one from the diagonal and add one to the same off-diagonal entry of *A*. This accomplishes the same thing with slightly different logic based on what I know how to do in *R*; here we start with a matrix of all zeros except ones where the log odds are missing and then figure out what the diagonal should be.

Block 2: Taking Care of Cells Missing from the Matrix of Log Odds Y

```
Build_A <- function (L, Y) {
  A = ifelse (Y,0,1)                  # Mark missing cells (this includes diagonal)
  diag(A) = L – (rowSums(A) – 1)      # Fix the diagonal (now every row sums to L)
 return (A) }
```

We can tweak the first block of code a little to take care of empty cells. This is pretty much the heart of the pair-wise method for estimating logit difficulties. With this and an R-interpreter, you could do it. However any functional, self-respecting, self-contained package would surround this core with several hundred lines of code to take care of the housekeeping to find and interpret the data and to communicate with you.

Block 3: More General Code Taking Care of Missing Cells

```
R = matrix (0, L, L)                    # Define and zero an LxL matrix
for (x in data)                         # Loop through people
    {R = R + ((1−x) %o% x)}             # Outer product of vectors creates square
Y = ifelse ((t(R)*R), log (t(R) / R), 0) # Log Odds (ji) over (ij)
    A = Build_A (L, Y)                  # Create coefficient matrix with empty cells
d = solve (A, rowSums(Y))               # Solve equations simultaneously
```

There is one gaping hole hidden in the rather innocuous expression, *for (x in data)*, which will probably keeping you from actually using this code. The vector *x* is the scored, zero-one item responses for one person. The object *data* presumably holds all the response vectors for everyone in the sample. The idea is to retrieve one response vector at a time, add it into the counts matrix *R* in the appropriate manner, until we've worked our way through everyone. I'm not going to tackle how to construct *data* today. What I will do is skip ahead to the fourth line and show you some actual data.

Table 1: Table of Count Matrix *R* for Five Multiple Choice Items

Counts	MC.1	MC.2	MC.3	MC.4	MC.5
MC.1	0	35	58	45	33
MC.2	280	0	240	196	170
MC.3	112	49	0	83	58
MC.4	171	77	155	0	99
MC.5	253	145	224	193	0

Table 1 is the actual counts for part of a real assessment. The entries in the table are the number of times the row item was missed and the column item was passed. The table is complete (i.e., all non-zeros except for the diagonal). Table 2 is the log odds computed from Table 1; *e.g., log (280/35) = 2.079* indicating item *2* is about two logits harder than item *1*. Because the table is complete, we don't really need the *A*-matrix of coefficients to get difficulty estimates; just add across each row and divide by five.

Table 2: Table of Log Odds *Y* for Five Multiple Choice Items

Log Odds	MC.1	MC.2	MC.3	MC.4	MC.5	Logit
MC.1	0	-2.079	-0.658	-1.335	-2.037	-1.222
MC.2	2.079	0	1.589	0.934	0.159	0.952
MC.3	0.658	-1.589	0	-0.625	-1.351	-0.581
MC.4	1.335	-0.934	0.625	0	-0.668	0.072
MC.5	2.037	-0.159	1.351	0.668	0	0.779

This brings me to the true elegance of the algorithm in Block 3. When we build the response vector *x* correctly (a rather significant qualification,) we can use exactly the same algorithm that we have been using for dichotomous items to handle polytomous items as well. So far, with zero-one items, the

response vector was a string of zeros and ones and the vector's length was the maximum possible score, which is also the number of items. We can coerce partial credit responses into the same format.

If, for example, we have a constructed response item with four categories, there are three thresholds and the maximum possible score is three. With four categories, we can parse the person's response into three non-independent items. There are four allowable response patterns, which not coincidentally, happen to be the four Guttman patterns: (000), (100), (110), and (111), which correspond to the four observable scores: 0, 1, 2, and 3. All we need to do to make our algorithm work is replace the observed zero-to-three polytomous score with the corresponding zero-one Guttman pattern.

Response	CR.1-2	CR.1-2	CR.1-3
0	0	0	0
1	1	0	0
2	1	1	0
3	1	1	1

If for example, the person's response vector for the five MC and one CR was (101102), the new vector will be (10110110). The person's total score hasn't changed but we know have a response vector of all ones and zeros of length equal to the maximum possible score, which is the number of thresholds, which is greater than the number of items. With all dichotomous items, the length was also the maximum possible score and the number of thresholds but that was also the number of items. With the reconstructed response vectors, we can now naively apply the same algorithm and receive in return the logit difficulty for each threshold.

Here are some more numbers to make it a little less obscure.

Table 3: Table of Counts for Five Multiple Choice Items and One Constructed Response

Counts	MC.1	MC.2	MC.3	MC.4	MC.5	CR.1-1	CR.1-2	CR.1-3
MC.1	0	35	58	45	33	36	70	4
MC.2	280	0	240	196	170	91	234	21
MC.3	112	49	0	83	58	52	98	14
MC.4	171	77	155	0	99	59	162	12
MC.5	253	145	224	193	0	74	225	25
CR.1-1	14	5	14	11	8	0	0	0
CR.1-2	101	46	85	78	63	137	0	0
CR.1-3	432	268	404	340	277	639	502	0

The upper left corner is the same as we had earlier but I have now added one three-threshold item. Because we are restricted to the Guttman patterns, part of the lower right is missing: e.g., you cannot pass item *CR.1-2* without passing *CR.1-1*, or put another way, we cannot observe non-Guttman response patterns like *(0, 1, 0)*.

Table 4: Table of Log Odds *Y* for Five Multiple Choice Items and One Constructed Response

Log Odds	MC.1	MC.2	MC.3	MC.4	MC.5	CR.1-1	CR.1-2	CR.1-3	Sum	Mean
MC.1	0	-2.079	-0.658	-1.335	-2.037	0.944	-0.367	-4.682	-10.214	-1.277
MC.2	2.079	0	1.589	0.934	0.159	2.901	1.627	-2.546	6.743	0.843
MC.3	0.658	-1.589	0	-0.625	-1.351	1.312	0.142	-3.362	-4.814	-0.602
MC.4	1.335	-0.934	0.625	0	-0.668	1.680	0.731	-3.344	-0.576	-0.072
MC.5	2.037	-0.159	1.351	0.668	0	2.225	1.273	-2.405	4.989	0.624
CR.1-1	-0.944	-2.901	-1.312	-1.680	-2.225	0	0	0	-9.062	-1.133
CR.1-2	0.367	-1.627	-0.142	-0.731	-1.273	0	0	0	-3.406	-0.426
CR.1-3	4.682	2.546	3.362	3.344	2.405	0	0	0	16.340	2.043

Moving to the matrix of log odds, we have even more holes. The table includes the row sums, which we will need, and the row means, which are almost meaningless. The empty section of the logs odds does make it obvious that the constructed response thresholds are estimated from their relationship to the multiple choice items, not from anything internal to the constructed response itself.

The A-matrix of coefficients (Table 5) is now useful. The rows define the simultaneous equations to be solved. For the multiple choice, we can still just use the row means because those rows are complete. The logit difficulties in the final column are slightly different than the row means we got when working just with the five multiple choice for two reasons: the logits are now centered on the eight thresholds rather than the five difficulties, and we have added in some more data from the constructed response.

Table 5: Coefficient Matrix *A* for Five Multiple Choice Items and One Constructed Response

A	MC.1	MC.2	*MC.3*	*MC.4*	MC.5	CR.1-1	CR.1-2	CR.1-3	Sum	Logit
MC.1	8	0	0	0	0	0	0	0	-10.214	-1.277
MC.2	0	8	0	0	0	0	0	0	6.743	0.843
MC.3	0	0	8	0	0	0	0	0	-4.814	-0.602
MC.4	0	0	0	8	0	0	0	0	-0.576	-0.072
MC.5	0	0	0	0	8	0	0	0	4.989	0.624
CR.1-1	0	0	0	0	0	6	1	1	-9.062	-1.909
CR.1-2	0	0	0	0	0	1	6	1	-3.406	-0.778
CR.1-3	0	0	0	0	0	1	1	6	16.340	3.171

This is not intended to be a R primer so much as an alternative way to show some algebra and arithmetic. I have found the R language to be a convenient tool for doing matrix operations, the R packages to be powerful tools for many perhaps most complex analyses, and the R documentation to be almost impenetrable. Mine is probably no better.