Running head: LOGICAL RULES

Supplementary Materials: Numerical predictions for serial, parallel and coactive logical rule-based models of categorization response time

Daniel R. Little

The University of Melbourne

Daniel R. Little

Psychological Sciences

The University of Melbourne

Parkville VIC 3053

e-mail: daniel.little@unimelb.edu.au

fax: + 61 3 9347 6618

Abstract

This document provides the supplementary materials to Little (2012; Numerical predictions for serial, parallel and coactive logical rule-based models of categorization response time). This supplement includes: i) MATLAB code for the coactive, serial and parallel models described in the main paper, and ii) a description of MATLAB code which can be used to estimate parameters for the models. MATLAB files can be downloaded from http://www.psych.unimelb.edu.au/research/labs/knowlab/documents/LogicalRuleModels.zip.

Supplementary Materials: Numerical predictions for serial, parallel and coactive logical rule-based models of categorization response time

This section contains MATLAB code for the coactive, serial and parallel logical-rule models.

```
function [pA, pN] = rw(varargin)
% function [pA, pN] = RW(M, boundaries, N, driftdist, startdist, startdistparms)
% [pUPPER, pN] = RW(M, BOUNDS, N) takes inputs M, BOUNDS, N and returns pUPPER,
% the probability of a random walk terminating at the UPPER boundary, and pN, the
% probability of a random walk terminating at step N.
%
% M is a scalar, which gives the average probability of taking a
% step towards the UPPER boundary (1 — M is the average probability of
% taking a step towards the LOWER boundary).
%
% BOUNDS, is a vector of UPPER and LOWER boundary positions. NUMEL(BOUNDS)
% should equal 2. Both BOUNDS should be positive; however, the function
% will check that BOUNDS(2) — the LOWER bound — is positive and negate it
% if necessary.
%
% N is a vector of step indexes to compute probabilities.
%
% Default parameters are: M = .5, BOUNDS = [10 10], N = 3000

%% Set up input
numvarargs = length(varargin);
if numvarargs > 7
    error('variableStartDrift_randomwalk:TooManyInputs', ...
        'requires at most 7 optional inputs');
end

```

```
% set defaults for optional inputs
optargs = {.5, [10 10], 3000};
% skip any new inputs if they are empty
newVals = cellfun(@(x) ¬isempty(x), varargin);
% now put these defaults into the valuesToUse cell array, and overwrite the ones
% specified in varargin.
optargs(newVals) = varargin(newVals);
% Place optional args in memorable variable names
[M, boundaries, N] = optargs{:};

%%
% Starting location = 0, so center boundary positions
A = boundaries(1);
B = boundaries(2);

A = ceil(A);        % Upper boundary
if sign(B) > 0
    B = -ceil(B);   % Lower boundary - if B is positive, round up to the
% nearest integer and negate
else
    B = floor(B);   % If B is already negative, round down to the
% nearest negative integer
end

m = (A - B) - 1;    % Number of transient steps

p = M;
q = 1 - p;
probp = 1;

i = -B; % Index row of starting position

% Variable starting point
Z = zeros(1, m);
Z(i) = 1;
Zidx = find(Z > 0);

```

```
64 %% Compute probability of ending at boundary B and A
65 pN = 0; pA = 0;
66 dpA = nan(1, numel(p));
67 pAA = nan(1, numel(Zidx));
68
69 zbounds = [m - Zidx' + 1, Zidx']; % Upper and Lower boundaries for sampling
70
71 for idx = 1:numel(Zidx)
72     startingsteps(1) = zbounds(idx, 1) - 1;
73     startingsteps(2) = zbounds(idx, 2) - 1;
74     Nsteps{1} = startingsteps(1):max(N); % Number of steps for which
75 % to calculate probabilities
76     Nsteps{2} = startingsteps(2):max(N);
77     % This should range from the minimum finishing time to maxN
78
79     % Probability of ending at the upper boundary
80     dpA(p == q) = Zidx(idx)./(m + 1);
81     % Probability of ending at the upper boundary
82     dpA(p ≠ q) = (1 - (q(p ≠ q)./p(p ≠ q)).^Zidx(idx))./...
83         (1 - (q(p ≠ q)./p(p ≠ q)).^(m+1));
84     pAA(idx) =  probp * dpA';
85
86     %% Compute probability of ending after N steps
87     % Compute first part of the function
88     % This gives a matrix of computations that is numel(p) x numel(Nsteps)
89     f1_A = repmat(-log(m + 1) + (Nsteps{1} + 1) * log(2), numel(p), 1) +...
90         (((Nsteps{1} + 1)/2)' * log(p .* q))' +...
91 repmat((((m + 1 - Zidx(idx))/2) * log(p./q))', 1, numel(Nsteps{1}));
92
93     f1_B = repmat(-log(m + 1) + (Nsteps{2} + 1) * log(2), numel(p), 1) +...
94         (((Nsteps{2} + 1)/2)' * log(p .* q))' +...
95 repmat(((Zidx(idx)/2) * log(q./p))', 1, numel(Nsteps{2}));
96
97     % Compute the second part of the function
98     f2_A = 0; f2_B = 0; % Initialize
99     for j = 1:m % Sum over each transient step
100         f2_A = f2_A + cos((pi * j)/(m + 1)).^Nsteps{1} *...
```

```
sin((pi * j * (m + 1 - Zidx(idx)))/(m + 1)) *...
sin((pi * j)/(m + 1));
        f2_B = f2_B + cos((pi * j)/(m + 1)).^Nsteps{2} *...
sin((pi * j * Zidx(idx))/(m + 1)) *...
sin((pi * j)/(m + 1));
    end

    % Combine parts 1 and 2
    pN_A = exp(f1_A) .* repmat(f2_A, numel(p), 1);
    pN_B = exp(f1_B) .* repmat(f2_B, numel(p), 1);

    % Note you need to add one extra zero for the step to the absorbing boundary
    % Keep RT distributions at each boundary separated
    pN_A = [zeros(numel(p), startingsteps(1)+1) pN_A];
    pN_B = [zeros(numel(p), startingsteps(2)+1) pN_B];
    zpN(:,:,idx) = [probp * pN_A; probp * pN_B];

    % Accumulate for starting point
    pN = pN + Z(Zidx(idx)) * zpN(:,:,idx);
    pA = pA + Z(Zidx(idx)) * pAA(idx);
end
pN(pN < eps) = 0; % If the probability is less than machine precision, set to zero
```

```
% Logical Rule Model - Serial
function [pData, pCorrect, nRT, pRT, eRT] =
 RWserial(parms, correctBoundary, nondtdist, rtdata, selfterm)
% parms = [driftrates, boundaries, residuals, pXfirst, rwScale]
% driftrates - pStep to upper for each process
% boundaries - upper and lower positions (the self-terminating boundary
% should be the lower boundary)
% pXfirst    - probability that the process that process X is run before process Y
% rwScale    - scaling factor to change steps to msec

M = parms(1:2);                 % Mean drift rate
boundaries = parms(3:4);        % Criterion Upper/Lower
```

```
13 rwScale = parms(5);             % Scaling parameter for steps to msec
14 residuals = parms(6:7);         % Residual mean and variance
15 pXfirst = parms(8);
16
17 N = 0:3000;
18
19 nProcesses = numel(M); % Currently only set up for 2 processes (X & Y)
20 if nProcesses ≠ 2; error('RWserial not set up for nProcesses ≠ 2'); end
21
22 %% Preallocate probabilities for each process
23 ppA = zeros(1, nProcesses);             % Probability that the process
24 % ends at the upper boundary
25 ppN  = zeros(2, numel(N)+1, nProcesses); % N boundaries, N steps + 1, N processes
26
27 %% Process pidx
28 % ppA is 1 x nProcesses
29 % pN  is 2 (Upper, Lower) x N x nProcesses
30 for pidx = 1:nProcesses
31     [ppA(1,pidx), ppN(:,:,pidx)] = rw(M(pidx), boundaries, N);
32 end
33 % The upper/A boundary is exhaustive, the lower/B boundary is
34 % self-terminating
35
36 %% Normalize distributions
37 % Hack to avoid divide by zero
38 ppA(ppA == 0) = 1e-10;
39 ppA(ppA == 1) = 1 - 1e-10;
40
41 % First process probability of ending after N steps at the upper boundary
42 pN(1,:,1) = ppN(1,:,1)./ppA(1);
43 % First process probability of ending after N steps at the lower boundary
44 pN(2,:,1) = ppN(2,:,1)./(1-ppA(1));
45 % Second process probability of ending after N steps at the upper boundary
46 pN(1,:,2) = ppN(1,:,2)./ppA(2);
47 % Second process probability of ending after N steps at the lower boundary
48 pN(2,:,2) = ppN(2,:,2)./(1-ppA(2));
49
```

```
%% Outcome probabilities — currently only set up for nProcesses = 2
% Probability that all processes end at the upper boundary
pAA = prod(ppA);
% Probability that process one ends at the upper boundary
pAB = ppA(1) * (1 — ppA(2));
% Probability that process two ends at the upper boundary
pBA = (1 — ppA(1)) * ppA(2);
% Probability that both processes end at the lower boundary
pBB = prod(1 — ppA);

%% Convolutions of step probabilities for each processes
cAA = fftConv(pN(1, :, 1), pN(1, :, 2)); cAA = cAA(1:(numel(N)+1)); % pxN_A + pyN_A
cAB = fftConv(pN(1, :, 1), pN(2, :, 2)); cAB = cAB(1:(numel(N)+1)); % pxN_A + pyN_B
cBA = fftConv(pN(2, :, 1), pN(1, :, 2)); cBA = cBA(1:(numel(N)+1)); % pxN_B + pyN_A
cBB = fftConv(pN(2, :, 1), pN(2, :, 2)); cBB = cBB(1:(numel(N)+1)); % pxN_B + pyN_B

%% Distribution of decision times for serial model
if selfterm
    if correctBoundary == 2 % Self—termininating item
        ps  = pBB * (pXfirst * pN(2,:,1) + (1 — pXfirst) * pN(2,:,2)) +...
              pBA * (pXfirst * pN(2,:,1) + (1 — pXfirst) * cBA) +...
              pAB * (pXfirst * cAB + (1 — pXfirst) * pN(2,:,2));
              pCorrect = pBB + pBA + pAB;
    else % correctBoundary == 1 % Exhaustive item
        ps = pAA * cAA;
        pCorrect = pAA;
    end
else % Exhaustive
    if correctBoundary == 2 % Self—termininating item
        ps = pBB * cBB + pBA * cBA + pAB * cAB;
        pCorrect = pBB + pBA + pAB;
    else
        ps = pAA * cAA; % Exhaustive item
        pCorrect = pAA;
    end
end

```

```
87  %% Insert scaled steps into larger array
88  if rwScale < 1; rwScale = 1; end
89  scaledDTsteps = round((1:numel(ps)) * rwScale - rwScale);
90  dt = [(0:max(scaledDTsteps))', zeros(max(scaledDTsteps) + 1, 1)];
91  dt(ismember(dt(:,1), scaledDTsteps), 2) = ps;
92  % The first column is x, the second column is dt
93
94  %% Convolve decision and nondecision times
95  switch nondtdist
96      case 'uniform'
97          ndt = unifpdf(dt(:,1), residuals(1), residuals(2));
98      case 'lognormal' % Non-decision time is log normal
99          ndt = (1./dt(:,1)) .* normpdf(log(dt(:,1)), residuals(1), residuals(2));
100     case 'normal'
101         ndt = normpdf(dt(:,1), residuals(1), residuals(2));
102     case 'none'
103         ndt = ones(size(dt,1),1);
104         pRT = dt(:,2)';
105 end
106
107 if ¬strcmp(nondtdist, 'none')
108     ndt(1) = 0; % Set first ndt to zero, as log(0) is bad
109     pRT = fftConv(dt(:,2), ndt); % Do FFT convolution
110 end
111 nRT = ((1:numel(pRT)) - 1);  % Get new time index
112 pRT(pRT < eps) = 0;
113 pRT = pRT./pCorrect;
114 eRT = sum(nRT .* pRT); % Expected RT
115 cRT = cumsum(pRT);
116
117 %% Compute probability of the data
118 if exist('rtdata', 'var') == 1
119     pData = zeros(numel(rtdata, 1)); % Initialize vector
120 else
121     rtdata = [];
122     pData = [];
123 end
```

```
124 if  ¬isempty(rtdata)
125     for i = 1:numel(rtdata)
126         rtidx = find(rtdata(i) ≤ nRT, 1, 'first');
127         if ¬isempty(rtidx)
128             pData(1,i) = cRT(rtidx); % Find appropriate probabilities
129         else
130             pData(1,i) = NaN;
131         end
132     end
133 end
```

```
1  % Logical Rule Model — Parallel
2  function [pData, pCorrect, nRT, pRT, eRT] =
3  RWparallel(parms, correctBoundary, nondtdist, rtdata, selfterm)
4  % parms = [driftrates, boundaries, residuals, pXfirst, rwScale]
5  % driftrates — pStep to upper for each process
6  % boundaries — upper and lower positions (the self—terminating
7  % boundary should be the lower boundary)
8  % rwScale    — scaling factor to change steps to msec
9
10 M = parms(1:2);                 % Mean drift rate
11 boundaries = parms(3:4);        % Criterion Upper/Lower
12 rwScale = parms(5);             % Scaling parameter for steps to msec
13 residuals = parms(6:7);         % Residual mean and variance
14
15 N = 0:3000;
16
17 nProcesses = numel(M); % Currently only set up for 2 processes (X & Y)
18 if nProcesses ≠ 2; error('RWparallel not set up for nProcesses ≠ 2'); end
19
20
21 %% Preallocate probabilities for each process
22 ppA = zeros(1, nProcesses);             % Probability that the process ends at A
23 ppN  = zeros(2, numel(N)+1, nProcesses); % N boundaries, N steps + 1, N processes
24
```

```
%% Process pidx
% ppA is 1 x nProcesses
% pN  is 2 (Upper, Lower) x N x nProcesses
for pidx = 1:nProcesses
    [ppA(1,pidx), ppN(:,:,pidx)] = rw(M(pidx), boundaries, N);
end

%% Normalize distributions
% Hack to avoid divide by zero
ppA(ppA == 0) = 1e-10;
ppA(ppA == 1) = 1 - 1e-10;

pN(1,:,1) = ppN(1,:,1)./ppA(1);
pN(2,:,1) = ppN(2,:,1)./(1-ppA(1));
pN(1,:,2) = ppN(1,:,2)./ppA(2);
pN(2,:,2) = ppN(2,:,2)./(1-ppA(2));

%% Outcome probabilities - currently only set up for nProcesses = 2
% Probability that all processes end at the upper boundary
pAA = prod(ppA);
% Probability that process one ends at the upper boundary
pAB = ppA(1) * (1 - ppA(2));
% Probability that process two ends at the upper boundary
pBA = (1 - ppA(1)) * ppA(2);
% Probability that both processes end at the lower boundary
pBB = prod(1 - ppA);

%% Minimum distribution for pBB - when the correct boundary is the upper boundary
% When both processes hit the lower boundary the rt is the minimum rt
mincdf1 = ((1-cumsum(pN(2,:,1))) .* (1-cumsum(pN(2,:,2))));
minpdf1 = abs([0, (diff(mincdf1))]);

% Minimum distribution for pBB - when the correct boundary is the upper boundary
mincdf2 = ((1-cumsum(pN(1,:,1))) .* (1-cumsum(pN(1,:,2)))); % For the second comparison
minpdf2 = abs([0, (diff(mincdf2))]);

%% Maximum distribution for pAA for when the correct boundary is the lower boundary
```

```
maxcdf1 = ((cumsum(pN(2,:,1))) .* (cumsum(pN(2,:,2))));
maxpdf1 = abs([0, (diff(maxcdf1))]);

% Maximum distribution for pAA for when the correct boundary is the upper boundary
maxcdf2 = ((cumsum(pN(1,:,1))) .* (cumsum(pN(1,:,2))));
maxpdf2 = abs([0, (diff(maxcdf2))]);

%% Distribution of decision times for parallel model
if selfterm
    if correctBoundary == 2 % Self-termininating item
        ps = pBB * minpdf1 + pBA * pN(2,:,1) + pAB * pN(2,:,2);
        pCorrect = pBB + pBA + pAB;
    else
        ps = pAA * maxpdf2;
        pCorrect = pAA;
    end
else
    if correctBoundary == 1
        ps = maxpdf1;
        pCorrect = pBB + pBA + pAB;
    else
        ps = maxpdf2;
        pCorrect = pAA;
    end
end

%% Insert scaled steps into larger array
if rwScale < 1; rwScale = 1; end
scaledDTsteps = round((1:numel(ps)) * rwScale - rwScale);
dt = [(0:max(scaledDTsteps))', zeros(max(scaledDTsteps) + 1, 1)];
dt(ismember(dt(:,1), scaledDTsteps), 2) = ps;
% The first column is x, the second column is dt

%% Convolve decision and nondecision times
switch nondtdist
    case 'uniform'
        ndt = unifpdf(dt(:,1), residuals(1), residuals(2));
```

```
99      case 'lognormal' % Non-decision time is log normal
100         ndt = (1./dt(:,1)) .* normpdf(log(dt(:,1)), residuals(1), residuals(2));
101     case 'normal'
102         ndt = normpdf(dt(:,1), residuals(1), residuals(2));
103     case 'none'
104         ndt = ones(size(dt,1),1);
105         pRT = dt(:,2)';
106 end
107
108 if ~strcmp(nondtdist, 'none')
109     ndt(1) = 0; % Set first ndt to zero, as log(0) is bad
110     pRT = fftConv(dt(:,2), ndt); % Do FFT convolution
111 end
112 nRT = ((1:numel(pRT)) - 1);  % Get new time index
113 pRT(pRT < eps) = 0;
114 pRT = pRT./pCorrect;
115 eRT = sum(nRT .* pRT); % Expected RT
116 cRT = cumsum(pRT);
117
118 %% Compute probability of the data
119 if exist('rtdata', 'var') == 1
120     pData = zeros(numel(rtdata, 1)); % Initialize vector
121 else
122     rtdata = [];
123     pData = [];
124 end
125 if  ~isempty(rtdata)
126     for i = 1:numel(rtdata)
127         rtidx = find(rtdata(i) ≤ nRT, 1, 'first');
128         if ~isempty(rtidx)
129             pData(1,i) = cRT(rtidx); % Find appropriate probabilities
130         else
131             pData(1,i) = NaN;
132         end
133     end
134 end
```

```
% Logical Rule Model — Coactive
function [pData, pCorrect, nRT, pRT, eRT] =
RWcoactive(parms, correctBoundary, nondtdist, rtdata)
% [pDATA, pUPPER, pRT] = RWcoactive(PARMS, NONDTDIST, RTDATA) takes
% inputs PARMS, CORRECTBOUNDARY, NONDTDIST, and RTDATA and returns pDATA, the
% probability of RT given the parameters in PARMS, pA, the probability of the process
% ending at the upper boundary, and pRT, the full distribution of RT
% probabilities
%
% PARMS is a vector containing the following parameters in the following
% order:
%   1) M — Mean drift rate (or ALPHA if driftdist = 'beta')
%   3) BOUNDARIES — Upper and Lower Boundary Locations
%   4) RWSCALE — Scanling parameter to scale random walk to msec (must be > 1)
%   5) RESIDUALS — Log—normal residual mean and variance
%
% If you want to use the defaults for any of these parameters, just set
% them to be empty vectors (e.g., startdist = [])
%
% RTDATA is a vector of RTDATA in msecs

M = parms(1);                   % Mean drift rate
boundaries = parms(2:3);        % Criterion Upper/Lower
rwScale = parms(4);             % Scaling parameter for steps to msec
residuals = parms(5:6);         % Residual mean and variance

N = 0:3000;

[pCorrect, pN] = rw(M, boundaries, N);

%% Insert scaled steps into larger array (1 msec)
ps = pN(correctBoundary,:); % Take correct boundary only
if correctBoundary == 2; pCorrect = 1 — pCorrect; end

% Scale the decision time
scaledDTsteps = round((1:numel(ps)) * rwScale — rwScale);
```

```
% Make two columns of times and likelihoods
dt = [(0:max(scaledDTsteps))', zeros(max(scaledDTsteps) + 1, 1)];
% Insert the likelihoods at the appropriate time
dt(ismember(dt(:,1), scaledDTsteps), 2) = ps;

%% Convolve decision and nondecision times
switch nondtdist
    case 'uniform'
        ndt = unifpdf(dt(:,1), residuals(1), residuals(2));
    case 'lognormal' % Non-decision time is log normal
        ndt = (1./dt(:,1)) .* normpdf(log(dt(:,1)), residuals(1), residuals(2));
    case 'normal'
        ndt = normpdf(dt(:,1), residuals(1), residuals(2));
    case 'none'
        ndt = ones(size(dt,1),1);
        pRT = dt(:,2)';
end

if ~strcmp(nondtdist, 'none')
    ndt(1) = 0; % Set first ndt to zero, as log(0) is bad
    pRT = fftConv(dt(:,2), ndt); % Do FFT convolution
end
nRT = ((1:numel(pRT)) - 1);  % Get new time index
pRT(pRT < eps) = 0;
pRT = pRT./pCorrect;
eRT = sum(nRT .* pRT); % Expected RT

%% Compute probability of the data
if exist('rtdata', 'var') == 1
    pData = zeros(numel(rtdata, 1)); % Initialize vector
else
    rtdata = [];
    pData = [];
end
if  ~isempty(rtdata)
    for i = 1:numel(rtdata)
        rtidx = find(rtdata(i) ≤ nRT, 1, 'first');
```

```
74          if ¬isempty(rtidx)
75              pData(i,1) = pRT(rtidx); % Find appropriate probabilities
76          else
77              pData(i,1) = NaN;
78          end
79      end
80  end
```

```
1   % Compute drift rate (p(Step to A)) given dimension boundaries, means and variances
2   function [p] = DBT_driftrate(x, dx, sx, coactive)
3
4   if ¬coactive
5       if size(x, 1) > 1
6           dx = repmat(dx, size(x, 1), 1);
7           sx = repmat(sx, size(x, 1), 1);
8       end
9
10      zx = ((dx - x)./sx);
11      p  = normcdf(zx, 0, 1);
12
13      % Truncate drift rate
14      p(p<.00001) = .00001;
15      p(p>.99999) = .99999;
16
17  else
18
19      %   B | D
20      %  ———————
21      %   A | C
22
23      aRegion = [-inf -inf; dx(1) dx(2)];
24      bRegion = [-inf dx(2); dx(1)  inf];
25      cRegion = [dx(1) -inf;  inf dx(2)];
26      dRegion = [dx(1) dx(2);  inf  inf];
27
```

```
    for i = 1:size(x, 1)
        p(i,:) = mvncdf(aRegion(1,:), aRegion(2, :), x(i,:), diag(sx)) +...
                 mvncdf(bRegion(1,:), bRegion(2, :), x(i,:), diag(sx)) +...
                 mvncdf(cRegion(1,:), cRegion(2, :), x(i,:), diag(sx));
    end
end
```

Model Fitting Example

This section contains a brief description of a parameter estimation routine for fitting a serial logical rule models described in Little (2012; Numerical predictions for serial, parallel and coactive logical rule-based models of categorization response time).

The file 'fitSerial.m' calls the 'fminsearch' function, which is passed a function called 'ruleRTserial'. 'ruleRTserial' is a wrapper function which calls the function 'RWserial' shown in the first part of the supplementary materials. 'ruleRTserial' compares the predictions returned by 'RWserial' for a given set of parameters, compares these predictions to the data, and returns the estimated fit value.

The parameters, all of which are identifiable within the model, are estimated using the Nelder-Mead SIMPLEX algorithm (Nelder & Mead, 1965) using the fminsearch.m MATLAB function. Model predictions are compared to the data using a quantile-based maximum likelihood function (Heathcote, Brown, & Mewhort, 2002); the negative of the log of this likelihood function is minimized to find the best fitting parameters. Further details of the QMLE procedure can be found in Little, Nosofsky, and Denton (2011).

The data were generated by simulating RTs from a mixed-order serial model. Further information is provided in the comments within the MATLAB files.

References

Heathcote, A., Brown, S., & Mewhort, D. J. K. (2002). Quantile maximum likelihood estimation of response time distributions. *Psychonomic Bulletin & Review*, *9*, 394-401.

Little, D. R., Nosofsky, R., & Denton, S. E. (2011). Response time tests of logical rule-based models of categorization. *Journal of Experimental Psychology: Learning, Memory & Cognition*, *37*, 1-27.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *Computer Journal*, *7*, 308-313.

Author Note

Preparation of this article was facilitated by ARC Discovery Grant DP120103120.
Address correspondence to the author at Psychological Sciences, The University of Melbourne, Parkville VIC 3053, Australia. Electronic mail may be sent to daniel.little@unimelb.edu.au. Software available at http://www.psych.unimelb.edu.au/research/labs/knowlab/documents/LogicalRuleModels.zip.