Functional Models for Test Items

Annotated Analyses in Matlab

In this chapter, we did four things:

- First, we used an EM algorithm to estimate item response functions $P_i(\ )$ and their log-odds counterparts $W_i(\ )$.
- Then we used principal components analysis to study the variation in these functions.
- We followed this with an estimation of the difference in performance between men and women on selected items.
- Finally we calculated arc length as a measure of performance and looked at its tangent for selected items.

In these notes we will see how these results are computed using the Matlab versions of the FDA functions, as well as some special Matlab functions for test analysis.

In these notes, we use the following notation:

- *N*: The number of examinees, here 2115.
- *n*: The number of items, here 60.
- *Q*: The number of quadrature points (see below), here 21.

Before you begin, don't forget to add the path to the functional data analysis functions. On my system, this is achieved by the command

```
addpath('../fdaM')
```

Inputting the Data

The raw data for the male examinees on the ACT Math Test are stored as 2115 rows of 60 0's and 1's indicating success or failure on an item, respectively, in a text file called `actm.txt`. These data are input as follows:

```
nit =   60;  %  Number of items
nex = 2115;  %  Number of examinees

fid      = fopen('actm.txt','rt');
temp     = fscanf(fid, '%s');
ACTmtest = reshape(temp, [nit,nex])';
```

Note that the second argument `'%s'` of the function `fscanf` instructions Matlab to read these 0's and 1's in as characters rather than numbers. Storing a very large table like this as characters greatly reduces the amount of memory used to store it.

Fixing the Charting Variable and Setting Quadrature Points and Weights

First we set up the charting variable used to map out position along the space curve. We will follow psychometric tradition in requiring that this have a standard normal distribution. The computation, however, will work with Q equally spaced values of ranging between –3.3 and 3.3. These are called in the algorithm *quadrature points* $_q$, and are used to approximate the integral used to calculate expected or marginal likelihood.

```
nq       = 21;  %  number of quadrature points
thetamax = 3.3; thetamin = -3.3;  %  range of theta values
thetaq   = linspace(thetamin, thetamax, nq)';
thetarng = [thetamin, thetamax];
```

The computation of the integral also requires some *quadrature weights* w_q, one weight for each quadrature point. These weights permit us to approximate the integral as follows:

$$\int_{-\infty}^{\infty} f(q)dq \approx \sum_{q=1}^{Q} w_q f\left(q_q\right)$$

It is shown in Ramsay and Rossi (2001) that a good method for choosing these quadrature weights w_q is to make them work for B-spline test functions, which roughly have the same shape characteristics as the integrands *f( )* in the approximation above. The following code does this.

```
norder = 4;  %  Order of the B-spline
nbasis = nq + norder - 2;  %   Number of basis functions
wgtq   = gausswgtBS(thetaq, nbasis, norder);
```

The special function `gausswgtBS` can be found in the appendix at the end of this document.

The B-spline Basis for Approximating Log-Odds Functions

In this step we want to set up 11 B-spline basis functions of order 4, a fairly standard choice of basis. We use 11 functions so as to achieve sufficient flexibility in the estimated functions, and we use a roughness penalty to control their smoothness.

```
nbasis = 11;
norder = 4;
wbasis = create_bspline_basis(thetarng, nbasis);
```

We will repeatedly need the matrix of values of these basis functions at the quadrature points.

```
phimat = getbasismatrix(thetaq, wbasis);
```

In addition, since we intend to use a roughness penalty, we require the roughness penalty matrix. The penalty in this case is, as is fairly standard, on the size of the second derivative of the estimated functions.

```
Kmat = getbasispenalty(wbasis, 2);
```

Initializing the EM Algorithm

The EM algorithm is a method refining an initial estimate of the item response functions. The following commands set up the initial estimates of these functions, as well as the log-odds functions corresponding to the 21 quadrature points. The special function FirstStep is given in the Appendix.

```
P0    = FirstStep(ACTmtest, thetaq);
W0   = log(P0./(1-P0));
```

These initial estimates are usually quite rough, and by converting the log-odds values to a functional data object, we both smooth the initial rough values a little, and set up an initial estimate of the coefficients of the B-spline expansion for each item response function.

```
wfd0  = data2fd(W0, thetaq, wbasis); % log-odds functions coef0 =
getcoef(wfd0);               % coefficients
```

The following two commands also provide the discrete function values corresponding to the quadrature points. But they will now be smoother than those computed above.

```
W    = eval(wfd0, thetaq); % initial log-odds values
P    = 1./(1+exp(-W0));         % initial probability values
Q    = 1 – P;              % failure values
Wfd  = Wfd0;               % initial functions W
coef = coef0;              % initial coefficient values
```

In addition to initial estimates of the functions and coefficients, we also want to set some parameters for the algorithm itself.

```
lambda   = 1e-1;           % smoothing parameter lambda
penmat   = lambda.*Kmat;   % penalty matrix times lambda
iter     = 0;              % initialize iteration number
convtest = 1e-2;           % convergence criterion
itermax  = 60;             % maximum number of iterations
F        = 1e10;           % initialize function value
Fold     = F + 2*convtest; % initialize old function value
```

Running the EM Algorithm

Now we're ready to run the EM algorithm. The algorithm itself consistes of repetitions of two types of analysis: The E-step in which the expected or marginal likelihood is computed, and the M-step in which this marginal likelihood is maximized with respect to the coefficients determining the log-odds functions. The Matlab code uses a `while` loop, in which convergence

of the algorithm is tested each time the initial `while` command is executed. The loop is closed by an `end` command. Here is the initial `while` command; it checks for convergence and for the current iteration number being less than the maximum allowed.

```
while Fold - F > convtest & iter < itermax
```

The following statements are now inside of the loop. These display an initial heading on the first iteration, and update the iteration number and the old function value.

```
   if iter == 0
      disp('It.    -log L   penalty    F     Fold - F');
   end
   iter = iter + 1;
   Fold = F;
```

Here is the E step. The special function `Estep` is found in the Appendix.

```
  [N, CN, CP, L, CL] = Estep(ACTmtest, P, wgtq);
```

This step produces the following arrays:

- `N`: A *Q*-vector containing an estimation of the number of examinees associated with each quadrature point. These can be called the *pseudo-sample-sizes*.
- `CN`: An *n* by *Q* matrix containing an estimation of the frequencies of success at each quadrature point for each item. These can be called the *pseudo-frequencies*.
- `CP`: An *N* by *Q* matrix containing estimated conditional probabilities associated with each examinee's response pattern conditional on ability being at each quadrature point.
- `L`: An *N*-vector containing marginal likelihood values for each examinee.
- `CL`: An *N* by *Q* matrix of conditional likelihood values for each examinee and each quadrature point.

The ratios if the pseudo-frequencies in array CN to the pseudo-sample-sizes in array N are themselves estimates of the probability of successes for *pseudo-examinees* whose ability is given by the quadrature points. We can call these values *pseudo-probabilities*.

These commands compute the current function value, which is the negative of the total marginal likelihood plus the size of the roughness penalty.

```
  %  compute penalized negative sum of marginal likelihoods
  logL = sum(log(L));
  pen  = sum(diag(coef' * penmat * coef));
  F = -logL + pen;
```

Now we print out the current results.

```
  fprintf('%g  ', [iter, -logL, pen, F, Fold - F]);
  fprintf('\n');
```

Here comes the M-step. It outputs the updated coefficient matrix optimizing the total marginal likelihood. The special function `Mstep` is given in the Appendix.

```
  coef = Mstep(CN, N, P, coef, phimat, penmat);
```

The last task in the loop is to update the *q* by *n* matrix of success probabilities associated with each quadrature point.

```
  P = 1./(1+exp(-phimat * coef));
```

The loop closes with and `end` statement.

```
End
```

For these data convergence was achieved in 30 iterations, and this took less than a minute on a 700 mherz personal computer running under Windows 98.

Our final act is to create a functional data object for the final log-odds functions.

```
Wfd = putcoef(Wfd, coef);
```

Displaying the Results

These commands display all of the 60 log-odds functions.

```
plot(Wfd)
xlabel('\fontsize{16} \theta')
ylabel('\fontsize{16} W(\theta)')
```

These commands display for each item

- The probabilities of success as a solid curve.
- The pseudo-probabilities, the ratios of estimated frequencies at quadrature points in array `CN` to the estimated sample sizes at quadrature points in array `N`, as points.
- The initial probabilities in array `P0` as a dashed curve.

```
itemindex = 1:nit;
for j = itemindex
  plot(thetaq, (CN(j,:)./N)', 'o',  ...
       thetaq, P(:,j),        'b-', ...
       thetaq, P0(:,j),       'g--')
  axis([thetamin, thetamax, 0, 1])
  xlabel('\fontsize{16} \theta')
  ylabel('\fontsize{16} P(\theta)')
  title(['\fontsize{16} Item ',num2str(j)])
  pause
end
```

What do we see?

- The item response functions in array `P` tend to track the pseudo-data fairly closely, but are noticeably more smooth.
- Both the pseudo-probabilities in `CN(j,:)./N` and the curve values are much smoother than the initial curve estimates in array `P0`.
- Both the curve estimatesand the pseudo-probabilities can do disconcerting things at the extremes of ability, where there are few examinees and therefore few actual responses to

determine the curves. More smoothing would probably not help this. But see Ramsay and Rossi (2001) for a more sophisticated roughness penalty that cures this problem.

We might also like to see what the estimated distribution of abilities is. The EM algorithm assumes that the abilities have a standard normal distribution, but the data may show something different. To look at this, we use the following code to estimate the probability density function and to display it.

```
pdf   = wgtq'.*sum(CP);
pdf   = pdf./sum(pdf');
plot(thetaq,pdf)
xlabel('\fontsize{16} \theta')
ylabel('\fontsize{16} Density')
title('\fontsize{16} Probability Density Function for Trait Score
Values')
```

Actually, it does really look pretty standard normal.

The following code plots the space curve shown in Figure 1 using the 21 points corresponding to the quadrature points.

```
plot3(P(:,1), P(:,9), P(:,59), 'o-')
axis([0,1,0,1,0,1])
xlabel('\fontsize{16} Item 1')
ylabel('\fontsize{16} Item 9')
zlabel('\fontsize{16} Item 59')
grid on
```

Displaying the Results with Arc Length

We saw that there are interesting arguments for replacing the values of by arc length *s* as a charting variable. The first step is to compute the derivatives of the item response functions $P_i(\ )$ with respect to at each quadrature point.

```
DWfdmat = eval(Wfd,thetaq,1);
DPmat   = P.*(1-P).*DWfdmat;
```

Next we compute the integrand in (4.3), namely the norm of the gradient vector, evaluated again at each quadrature point.

```
DPsqr  = sum((DPmat').^2);
DPnorm = sqrt(DPsqr)';
```

Here we compute the arc length measure of the length of the whole space curve. Note that the last command approximates the integral in (4.3) by using the trapezoidal rule, a basic but highly effective method for the numerical approximation of an integral.

```
Smax = (thetaq(2) - thetaq(1)).*(sum(DPnorm) - ...
            0.5.*(DPnorm(1) + Dpnorm(nq)));
```

Now we are ready to compute arc length values corresponding to each quadrature point. Here we use the function monfn that evaluates monotone functions to compute these. The arc length values corresponding to the quadrature points are in vector Svec.

```
Msfd = data2fd(log(DPnorm), thetaq, wbasis);
Svec = monfn(thetaq,Msfd);
Svec = Smax.*Svec./max(Svec);
```

Here we plot arc length as a function of slope, as in Figure 4.7.

```
plot(thetaq, Svec)
xlabel('\fontsize{16} \theta')
ylabel('\fontsize{16} Arc Length s')
```

We will also want to express the log-odds function as a function of arc length *s* rather than . To do this we want to create a new basis, since arc length is defined over a different range of values than is .

```
Wmat    = eval(Wfd, thetaq);
Wsbasis = create_bspline_basis([0,Smax], 16);
Wsfd    = data2fd(Wmat, Svec, Wsbasis);
```

One reason for switching to arc length measure is the fact that the tangent vector has unit length, and therefore provides a natural measure of item discriminability. Here we compute the tangent vector functions.

```
DsPmat = DPmat./(DPnorm*ones(1,nit));
```

Principal Components Analysis of Log-Odds Functions

The principal components analysis of the log-odds functions $W_i(\ )$ is now a straightforward application of the functional version of PCA. These three commands carry out the PCA and apply a Varimax rotation to the resulting principal components or harmonics.

```
nharm   = 4;
Wpcastr = pca(Wfd, nharm);
Wpcastr = varmx_pca(Wpcastr);
```

Next we evaluate the harmonics of the log-odds functions at the quadrature points.

```
Wharmmat = eval(Wpcastr.harmfd, thetaq);
```

We also need the values of the mean log-odds function and the mean item response function.

```
Wfdmean  = mean(Wfd);
Wmean    = eval(Wfdmean, thetaq);
Wmeanmat = Wmean*ones(1,4);
```

```
Pmean    = exp(Wmean)./(1 + exp(Wmean));
```

Next we add and subtract a judicious amount of each harmonic to the mean log-odds function, and convert the results to the corresponding item response functions.

```
Wconst = ones(nq,1)*[2, 1, .5, .5];
Wmatp  = Wmeanmat +  Wconst.*Wharmmat;
Wmatm  = Wmeanmat -  Wconst.*Wharmmat;
Pharp  = exp(Wmatp)./(1+exp(Wmatp));
Pharm  = exp(Wmatm)./(1+exp(Wmatm));
```

Finally, we plot the results.

```
titlestr = ['  I: 30%'; ' II: 31%'; ...
            'III: 17%'; ' IV: 19%'];
for j=1:nharm
  subplot(2,2,j)
  plot(thetaq, Pmean, '--')
  axis([thetamin, thetamax, 0, 1])
  text(thetaq-.1, Pharp(:,j), '+')
  text(thetaq-.1, Pharm(:,j), '-')
  title(['\fontsize{12} Harmonic ',titlestr(j,:)])
end
```

Appendix: Special Matlab Functions

Function `FirstStep`

```
function P0 = FirstStep(dichtest, thetaq)
%  initialize EM algorithm by finding initial probabilities
%  Arguments:
%    dichtest ... a nex by nit matrix of binary item scores
%    thetaq   ... set of quadrature points
%  Return:
%    P0 ... a nq by nit matrix of proportions

[nex,nit] = size(dichtest);  % compute no. examinees and items
nq = length(thetaq);         %  number of trait values

% we want to make a histogram, with each bin centered
%   on a trait value.  First construct nq+1 boundaries for bars

bounds = zeros(nq+1,1);
bounds(   1) = -1e10;
```

```
bounds(nq+1) =  1e10;
for q = 2:nq
  bounds(q) = (thetaq(q-1)+thetaq(q))/2;
end

%  for each examinee, compute the index of the bar or bin
%    containing his quantile value

qscore = norminv((1:nex)./(nex+1), 0, 1);

%  get indices of bins corresponding to quantiles

binindex = zeros(nex,1);
for i=1:nq
  index = (qscore <= bounds(i+1) & qscore > bounds(i));
  binindex(index) = i;
end

%  Get preliminary estimates of IRF's at thetaq values
%  note:  the data play a role here only in terms of the
%  sorting index array, sortindex.  This array sorts the rows of
%  the dichotomous response matrix according to the rank of
%  the number right scores.

%  compute scores on the test
score = zeros(nex,1);
for i=1:nex
  temp = double(dichtest(i,:))-48;
  %temp = temp(temp ~= 2);
  %score(i) = (sum(temp)/length(temp))*nit;
  score(i) = sum(temp);
end
%  sscore are the sorted scores, sortindex the indices that
%     sort vector score.  A random normal deviate, mean 0,
%     std. dev. .01 is added to each score before sorting
%     to sort tied values in random order.
[sscore,sortindex] = sort(score+0.01.*randn(nex,1));

%  for each item, compute proportion of examinees in each bin
%  that pass the test using function bin

biny = zeros(nq,nit);
for j = 1:nit
   temp = double(dichtest(sortindex,j))-48;
   biny(:,j) = bin(nq, temp, binindex);
end

%  Function bound replaces probability values in biny by
%  1/(2*NEX) if lower, or by 1 - 1/(2*NEX) if higher.

P0 = bound(biny, nex);
```

Function `bin`

```
function binx = bin(nbin, rgt, binindex, meanwrd)
  %  Bins values in vector RGT into bins indicated in vector
  %      BININDEX
  %  If MEANWRD is T, the value in the bin is the of the RGT
  %      values in the bin, otherwise it is the sum.
  if nargin < 4, meanwrd = 1; end
  binx = zeros(nbin,1);
  binind = unique(binindex);

  if meanwrd
    for i = 1:nbin
      temp = rgt(binindex==i);
      if length(temp) > 0, binx(i) = mean(temp); end
    end
  else
    for i = 1:nbin
      binx(i) =  sum(rgt(binindex==i));
    end
  end
```

Function `bound`

```
function pmat = bound(pmat, nex)
   %  replaces probability values in PMAT by 1/(2*NEX) if lower,
   %  or by 1 - 1/(2*NEX) if higher.
   delta = 1/(2*nex);
   pmatdim = size(pmat);
   for j=1:pmatdim(2)
     index = pmat(:,j) <   delta;
     if any(index), pmat(index,j) =     delta; end
     index = pmat(:,j) > 1-delta;
     if any(index), pmat(index,j) = 1 - delta;  end
   end
```

Function `Estep`

```
function [N, CN, CP, L, CL] = Estep(dichtest, P, wgtq)
%  The E step of the EM algorithm
%  dichtest ... N by n matrix of binary item scores
%  P        ... Q by n matrix of probabilities
%  wgtq     ... Q by 1 vector of quadrature weights

% get number of examinees and number of items
[nex, nit] = size(dichtest);
Q = length(wgtq);  %  get number of quadrature weights
%  Compute:
%  CL:  N by Q matrix of conditional likelihoods,
```

```
%  L:   N marginal likelihoods,
%  CP:  N by Q matrix of conditional probabilities, and
%  CN:  n by Q matrix of conditional pseudo-frequencies
CL = zeros(nex,Q);
CP = CL;
CN = zeros(nit,Q);
L  = zeros(nex,1);
logP   = log(P);
log1mP = log(1 - P);
for i=1:nex
  temp = (double(dichtest(i,:))-48)';
  %notmiss = (temp ~= 2);
  %temp = temp(notmiss);
  %CL(i,:) = exp(logP(:,notmiss)*temp + ...
              log1mP(:,notmiss)*(1-temp))';
  CL(i,:) = exp(logP*temp + log1mP*(1-temp))';
  %plot(thetaq,CL(i,:)');
  L(i) = CL(i,:)*wgtq;
  CP(i,:) = wgtq'.*CL(i,:)./L(i);
  %CN(notmiss,:) = CN(notmiss,:) + temp*CP(i,:);
  CN = CN + temp*CP(i,:);
  %pause
end
%  Compute Q marginal pseudo-frequencies
N = sum(CP);
```

Function `Mstep`

```
function newcoef = Mstep(CN, N, P, coef, phimat, penmat)
%  M-step
%  for each item in turn, minimize criterion Fj with respect to
%    values of coefficients cj
%  CN     ... nit by Q matrix of conditional pseudo frequencies
%  N      ... vector of Q marginal pseudo frequencies
%  P      ... Q by nit matrix of probabilities
%  coef   ... K by nit matrix of coefficients for basis functions
%  phimat ... Q by K matrix of basis function values
%  penmat ... K by K matrix for penalizing coefficient roughness
nit = size(P,2);
newcoef = coef;
iterlim = 10;
for j=1:nit
    cj   = coef(:,j);             % initial coefficients
    pj   = P(:,j); qj = 1-P(:,j); % initial probabilities
    wj   = diag(pj.*qj.*N');      % initial weights
    CNj  = CN(j,:)';              % "Data" for this item
    resj = CNj - pj.*N';          % initial residuals
    %  initial function value
    Fj = -sum(CNj.*log(pj)+(N'-CNj).*log(qj)) + cj'*penmat*cj;
    %  gradient vector
    Gmat = -phimat' * resj        + 2.*penmat*cj;
```

```
    %  Hessian matrix
    Hmat =  phimat' * wj * phimat + 2.*penmat;
    %  search direction
    delta = Hmat \ Gmat;
    %  initial step size along direction
    alpha = 1;
    Fjnew = Fj + 1;
    %  take step, and if new function value less than old, quit
    %  otherwise halve step and try again.
    %  in any case, stop when step size gets too small
    iter = 0;
    while (Fjnew > Fj & iter <= iterlim)
       iter = iter + 1;
       cjnew = cj - alpha .* delta; % update coefficients
       %  Don't let coefficients get too large in either
       %     direction
       cjnew(cjnew > 20) = 20; cjnew(cjnew < -20) = -20;
       % update probabilities
       pjnew = 1./(1+exp(-phimat*cjnew)); qjnew = 1 - pjnew;
       % compute new function value
       Fjnew = -sum(CNj.*log(pjnew)+(N'-CNj).*log(qjnew)) + ...
                 cjnew'*penmat*cjnew;
       %fprintf('%g ', [j, iter, Fjnew, Fj]); fprintf('\n');
       alpha = alpha/2; % halve step size
    end
    newcoef(:,j) = cjnew;   % replace old coefficients by new
ones
end
```

Function `gausswgt3`

```
function [wgt, integvec] = gausswgt3(thetaq, nbasis, norder)
%  Quadrature weights for B-spline test functions

%  set up fine mesh for estimating integrals

nq        = length(thetaq);
thetarng  = [min(thetaq), max(thetaq)];
delta     = 1e-3;
thetafine = min(thetaq):delta:max(thetaq);
nfine     = length(thetafine);

%  set up spline basis

norder   = 4;
nbasis   = nq + norder - 2;
basisobj = create_bspline_basis(thetarng, nbasis);

basisnq  = getbasismatrix(thetaq,    basisobj);
basisfin = getbasismatrix(thetafine, basisobj);
```

```
%  compute std. normal density values

kernelfn = (exp(-thetafine.^2/2)/sqrt(2*pi))';

%  estimate integrals by trapezoidal rule

integvec = delta.*(basisfin'*kernelfn -
0.5.*(basisfin(1,:)'.*kernelfn(1) +
basisfin(nfine,:)'.*kernelfn(nfine)));

%  solve for weights giving least squares solution

wgt = basisnq'\integvec;
```