# Lecture 5: More Nuts and Bolts—FIML and Categorical Data MI

EPSY 6349: Modern Missing Data Analysis

Kyle M. Lang

Institute for Measurement, Methodology, Analysis & Policy
Texas Tech University
Lubbock, TX

September 29, 2015

## Outline

- Look at FIML in more depth
  - Show how to do ML and FIML (semi) by hand in R.
- Discuss MI for categorical variables.
  - Show a manual R example.

## FIML Conceptual Refresher

FIML is an ML estimation method that is robust to ignorable nonresponse.

FIML partitions the missing information out of the likelihood function so that the model is only estimated from the observed parts of the data.

After a minor alteration to the likelihood function, FIML reduces to simple ML estimation.

So, let's review ML estimation before moving forward.

## Maximum Likelihood Estimation Refresher

ML estimation simply finds the parameter values that are "most likely" to have given rise to the observed data.

- The *likelihood* function is just a probability density (or mass) function with the data treated as fixed and the parameters treated as random variables.
- Having such a framework allows us to ask: "Given that I've observed these data values, what parameter values most probably describe these data?"

## Maximum Likelihood Estimation Refresher

ML estimation simply finds the parameter values that are "most likely" to have given rise to the observed data.

- The *likelihood* function is just a probability density (or mass) function with the data treated as fixed and the parameters treated as random variables.
- Having such a framework allows us to ask: "Given that I've observed these data values, what parameter values most probably describe these data?"

ML estimation is usually employed when there is no closed form solution for the parameters we seek.

- This is why you don't usually see ML used to fit OLS regression models or ordinary ANOVAs.

After choosing a likelihood function, we iteratively optimize the function to produce the ML estimated parameters.

- In practice, we nearly always work with the natural logarithm of the likelihood function (i.e., the *loglikelihood*).

## Steps of ML

1. Choose a loglikelihood function $f(X|\boldsymbol{\theta})$ to describe the distribution of the data given the parameters.
2. Choose some initial estimate of $\boldsymbol{\theta}$, $\hat{\boldsymbol{\theta}}^{(0)}$.
3. Compute each row's value for the likelihood function parameterized by $\hat{\boldsymbol{\theta}}$, $f(x_n|\hat{\boldsymbol{\theta}})$.
4. Sum the individual loglikelihood value into the joint loglikelihood of the data: $LL = \sum_{n=1}^{N} f(x_n|\hat{\boldsymbol{\theta}})$.
5. Choose a "better" estimate of the parameters $\hat{\boldsymbol{\theta}}^{(1)}$ and repeat steps 3 and 4.
6. Repeat steps 3 – 5 until the change between $LL^{(i)}$ and $LL^{(i+1)}$ is negligible.
7. Take $\hat{\boldsymbol{\theta}}^{(i)}$ as your estimated parameters.

## ML Example

Recall the multivariate normal loglikelihood function:

$$LL_n = -\frac{P}{2}\ln(2\boldsymbol{\pi}) - \frac{1}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{Y}_n - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{Y}_n - \boldsymbol{\mu})$$

We can define this function manually in R:

```r
## Row i's contribution to the overall LL:
llVec[i] <- -0.5 * nVar * log(2 * pi) -
    0.5 * log(det(sigmaHat)) -
        0.5 * (inData[i, ] - muHat) %*%
            solve(sigmaHat) %*%
                matrix(inData[i, ] - muHat)
```

## ML Example

We can wrap the preceding code in an R function that takes a parameter vector and a data set and returns the total loglikelihood value for the data:

```r
## Complete data loglikelihood function:
myLL ← function(parVec, inData)
{
    ## Extract the parameter matrices:
    nVar ← ncol(inData)
    muHat ← parVec[1 : 3]
    sigmaHat ← matrix(parVec[4 : 12], ncol = 3)

    ## Compute the row-wise LogLikelihoods:
    llVec ← rep(NA, nrow(inData))
    for(i in 1 : nrow(inData)) {
        llVec[i] ← -0.5 * nVar * log(2 * pi) -
            0.5 * log(det(sigmaHat)) -
                0.5 * (inData[i, ] - muHat) %*%
                    solve(sigmaHat) %*%
                        matrix(inData[i, ] - muHat)
    }
    sum(llVec)# return the overall LL value
}
```

## ML Example I

The **optimx** package in R can numerically optimize arbitrary functions. We can use it to do ML by hand.

```r
parms ← list()
parms$nObs ← 1000
parms$meanVec ← c(1.0, 2.0, 3.0)# x, y, z
parms$corVec ← c(0.2, 0.5, 0.4)# xy, xz, yz
## Create some toy data:
simData ← simulateSimpleData(parms)
## Vector of starting values for the parameters:
startVec ← c(rep(0.5, 3),
              c(1.0, 0.3, 0.3,
                0.3, 1.0, 0.3,
                0.3, 0.3, 1.0))
## Use optimx() to numerically optimize the LL function:
llOpt ← optimx(startVec,
                fn = myLL,
                inData = as.matrix(simData),
                method = "L-BFGS-B",
                lower = 0.0,
                control = list(maximize = TRUE,
                    kkt = FALSE))
```

## ML Example II

```
Maximizing -- use negfn and neggr
```

```
## Get the optimize mean vector and covariance matrix:
muHatLL ← llOpt[1 : 3]
sigmaHatLL ← matrix(as.numeric(llOpt[4 : 12]), ncol = 3)
## Look at the results:
muHatLL
```

```
               p1       p2       p3
L-BFGS-B 1.033976 1.985203 3.009896
```

```
sigmaHatLL
```

```
          [,1]      [,2]      [,3]
[1,] 1.0125147 0.1805506 0.4952722
[2,] 0.1805501 0.9265573 0.3427037
[3,] 0.4952725 0.3427032 0.9519676
```

# ML Example III

```
muHatLL - colMeans(simData)
```

```
                   p1            p2            p3
L-BFGS-B 2.984267e-06 -7.713401e-06 -6.986144e-06
```

```
sigmaHatLL - cov(simData)
```

```
              x             y             z
x -0.0010179749 -0.0001840542 -0.0005330290
y -0.0001845228 -0.0009284928 -0.0003518237
z -0.0005327091 -0.0003522739 -0.0009734289
```

## FIML Example

We can do this same process for FIML estimation.

Recall the FIML loglikelihood function:

$$LL_{FIML,n} = -\frac{P}{2}\ln(2\boldsymbol{\pi}) - \frac{1}{2}\ln|\Sigma_n| - \frac{1}{2}(\mathbf{Y}_n - \boldsymbol{\mu}_n)^T \Sigma_n^{-1} (\mathbf{Y}_n - \boldsymbol{\mu}_n)$$

We can also define this version manually in R:

```r
## Subset mu, sigma, and inData to only work
## with the observed parts of inData:
isObs <- !is.na(inData[i, ])
nVar <- sum(isObs)
llVec[i] <- -0.5 * nVar * log(2 * pi) -
    0.5 * log(det(sigmaHat[isObs, isObs])) -
        0.5* (inData[i, isObs] - muHat[isObs]) %*%
            solve(sigmaHat[isObs, isObs]) %*%
                matrix(inData[i, isObs] - muHat[isObs])
```

## FIML Example

Construct a wrapper function to get the overall LL value:

```
## FIML loglikelihood function:
myFIML ← function(parVec, inData)
{
    ## Extract the parameter matrices:
    nVar ← ncol(inData)
    muHat ← parVec[1 : 3]
    sigmaHat ← matrix(parVec[4 : 12], ncol = 3)

    ## Compute the row-wise -2LogLikelihoods:
    llVec ← rep(NA, nrow(inData))
    for(i in 1 : nrow(inData)) {
        ## Subset mu, sigma, and inData to only work
        ## with the observed parts of inData:
        isObs ← !is.na(inData[i, ])
        nVar ← sum(isObs)

        llVec[i] ← -0.5 * nVar * log(2 * pi) -
            0.5 * log(det(sigmaHat[isObs, isObs])) -
                0.5 * (inData[i, isObs] - muHat[isObs]) %*%
                    solve(sigmaHat[isObs, isObs]) %*%
                        matrix(inData[i, isObs] - muHat[isObs])
    }
    sum(llVec) # return the overall LL value
}
```

## FIML Example I

And we can use **optimx** to numerically optimize the FIML objective:

```
parms$pm ← 0.3
parms$auxVar ← "z"
parms$incompVars ← "y"
parms$marType ← "lower"
# Impose missing data on the toy data:
missData ← imposeMissing(simData, parms)
## Use optimx() to numerically optimize the FIML objective:
fimlOpt ← optimx(startVec,
                  fn = myFIML,
                  inData = as.matrix(missData),
                  method = "L-BFGS-B",
                  lower = 0.0,
                  control = list(maximize = TRUE,
                      kkt = FALSE))
```

```
Maximizing -- use negfn and neggr
```

## FIML Example II

```
## Get the optimized mean vector and covariance matrix:
muHatFIML ← as.numeric(fimlOpt[1 : 3])
sigmaHatFIML ←
    matrix(as.numeric(fimlOpt[4 : 12]), ncol = 3)
## Look at the results:
muHatFIML
```

```
[1] 1.033963 1.945265 3.009926
```

```
sigmaHatFIML
```

```
          [,1]      [,2]      [,3]
[1,] 1.0125127 0.1896069 0.4953259
[2,] 0.1896089 0.9016940 0.3807157
[3,] 0.4953237 0.3807173 0.9519759
```

## FIML Example I

Just to make sure our results are plausible, we can do the same analysis using lavaan():

```
## Confirm the manual approach by using
## lavaan() to get the FIML estimates:
mod1 ← "
x ~~ y + z
y ~~ z
"
## Fit the model with lavaan():
out1 ← cfa(mod1, data = missData, missing = "fiml")
muHatFIML2 ← inspect(out1, "est")$nu
sigmaHatFIML2 ← inspect(out1, "theta")
```

## FIML Example II

```
## Same as what we got by hand :)
sigmaHatFIML - sigmaHatFIML2
```

```
  x y z
x 0 0 0
y 0 0 0
z 0 0 0
```

```
muHatFIML - muHatFIML2
```

```
  intrcp
x      0
y      0
z      0
```

## Categorical Variable MI

## Orienting Categorical Variable MI

As we've seen, MI is basically just a slightly more complicated flavor of out-of-sample prediction using regression.

- So far, we've only considered imputation using linear regression models.
- What do we need to modify to address categorical outcomes?
  - Can we use OLS for prediction with categorical DVs?

## Motivating the Generalized Linear Model

OLS regression is a type of *General Linear Model.*

- A normally distributed outcome variable is linearly associated with a set of predictor variables (which may be non-linear transformations of the observed predictors)

Generalized Linear Models take the form:

$$Y = \mathbf{X}\beta + \varepsilon$$
$$\text{with } \varepsilon = Y|\mathbf{X} \sim \mathrm{N}(\mu, \sigma^2)$$

If we have a binary outcome, for example, we can't simply assume it follows a normal distribution and model it as the linear combination of some predictors.

- Why not?

## Overview of the Generalized Linear Model

The *Generalized Linear Model* is an extension of the General Linear Model that allows the DV to follow non-normal distributions.

- Most importantly, the Generalized Linear Model allows for discrete distributions of the DV

Generalized Linear Models take the form:

$$g(\boldsymbol{\mu}_Y) = \mathbf{X}\boldsymbol{\beta}$$
$$\text{with } Y|\mathbf{X} \sim f(Y|\boldsymbol{\mu}, \phi)$$

Some terminology:

- $\mathbf{X}\boldsymbol{\beta}$ is called the *Systematic Component*
- $f(Y|\boldsymbol{\mu}, \phi)$ is called the *Random Component*
- $g(\cdot)$ is called the *Link Function*
  - $g^{-1}(\cdot)$ is sometimes called the *Activation Function.*

## Parts of the Generalized Linear Model

The systematic component is simply the weighted sum of the predictors

- Exactly the same as in OLS regression

## Parts of the Generalized Linear Model

The systematic component is simply the weighted sum of the predictors

- Exactly the same as in OLS regression

The random component is the (conditional) distribution of the outcome after accounting for the systematic component.

- In OLS regression, the systematic component is the normal distribution
- Logistic and Probit regression use the *Binomial Distribution* as their random component.

## Parts of the Generalized Linear Model

The link function "linearizes" the outcome variable, so that it can be modeled by the systematic component.

- OLS regression uses the so-called *Identity Link*:

$$g(\mu_Y) = \mu_Y.$$

- Logistic regression uses the *Logit Link*:

$$g(\mu_Y) = \ln\left\{\frac{P(y=1|\mathbf{X})}{(1-P(y=1|\mathbf{X}))}\right\},$$

  which represents the log-odds of success on Y.

- Probit regression uses the *Probit Link*:

$$\mu_Y = \Phi(\mathbf{X}\beta) \Rightarrow g(\mu_Y) = \Phi^{-1}(\mathbf{X}\beta),$$

  which returns the quantile on the standard normal CDF associated with each value of the systematic component.

## Latent Variable Representation of GLMs

GLMs are usually fit directly to the discrete data using marginal ML or iteratively reweighted least squares.

Many GLMs, however, can be formulated in terms of unobserved *Latent Response Indicators*.

- If we assume that our observed, discrete outcome is an imperfect coarsening of an underlying continuous variable, we can model the unobserved continuous variable.

Probit Regression already implicitly assumes a latent response indicator. It's latent formulation takes the form:

$$Y^* = \mathbf{X}\beta + \varepsilon$$

$$Y = \begin{cases} 1 & \text{when} \quad Y^* > 0 \\ 0 & \text{when} \quad Y^* \leq 0 \end{cases}$$

## Why do we care?

The latent variable representations of GLMs can dramatically simplify Bayesian modeling of some GLMs.

Due to the simple formulate shown above, we can construct the following Bayesian Probit Model:

$$Y^* \sim \begin{cases} \text{Trunc-N}^{+}(\mathbf{X}\boldsymbol{\beta},\ \mathbf{1}) & \text{when} \quad Y = 1 \\ \text{Trunc-N}^{-}(\mathbf{X}\boldsymbol{\beta},\ \mathbf{1}) & \text{when} \quad Y = 0 \end{cases}$$

$$\boldsymbol{\beta} \sim \text{N}\left((\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y^*,\ (\mathbf{X}^T\mathbf{X})^{-1}\right)$$

$$Y = \begin{cases} 1 & \text{when} \quad Y^* > 0 \\ 0 & \text{when} \quad Y^* \leq 0 \end{cases}$$

# Creating Imputations from the Latent Probit Model: Model Estimation

1. Start with an initial guess of $Y^{*(0)}$.
2. Regress $Y^{*(0)}$ onto $\mathbf{X}_{obs}$ to get $\hat{\beta}^{(0)} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y^{*(0)}$
3. Use $\hat{\beta}^{(0)}$ to parameterize $P(\beta | Y^*, \mathbf{X})$
4. Sample a new vector of coefficients $\hat{\beta}^{(1)}$ from $P(\beta | Y^*, \mathbf{X})$.
5. Use $\hat{\beta}^{(1)}$ to compute $\mathbf{X}\beta$ and parameterize $P(Y^* | \beta, \mathbf{X})$.
6. Sample $Y^{*(1)}$ from $P(Y^* | \beta, \mathbf{X})$.
7. Repeat steps 2–6 until the model converges.

## Creating Imputations from the Latent Probit Model: Drawing Imputations

1. Randomly sample $M$ sets of parameters $\boldsymbol{\beta}^{(m)}$ from the stationary $P(\boldsymbol{\beta} | Y^{*}, \mathbf{X})$
2. For each $\boldsymbol{\beta}^{(m)}$ compute $Y_{mis}^{*(m)} = \mathbf{X}_{mis} \boldsymbol{\beta}^{(m)}$
3. Calculate the predicted probabilities or success $P(Y = 1 | \mathbf{X}) = \Phi(Y^{*(m)})$ associated with each $Y^{*(m)}$
4. Compare the $P(Y = 1 | \mathbf{X})$ to a standard uniform variate $u_i$.
5. Create the imputations as follows:

$$Y_{imp} = \begin{cases} 1 & \text{when} \quad P(Y = 1 | \mathbf{X}) > u_i \\ 0 & \text{otherwise} \end{cases}$$

## Bayesian Probit Example I

```
nReps ← 5000
nImps ← 100
## Simulate some data:
simData ← simulateSimpleData(parms)
## Categorize Y:
simData[ , "y"] ← as.numeric(cut(simData[ , "y"], 2)) - 1
## Impose missing:
missData ← imposeMissing(simData, parms)
## Subset the data:
yMis ← missData[is.na(missData[ , "y"]), ]
yObs ← missData[!is.na(missData[ , "y"]), ]
## Get some useful summaries:
ySuccess ← yObs[ , "y"] == 1
predMat ← as.matrix(cbind(1, yObs[ , c("x", "z")]))
nObs ← nrow(yObs)
## Start the Gibbs sampled parameters:
yStar ← rnorm(nObs)
startFit ← lm.fit(y = yStar, x = predMat)
coefs ← startFit$coef
yHats ← startFit$fitted
coefVar ← solve(crossprod(predMat))
```

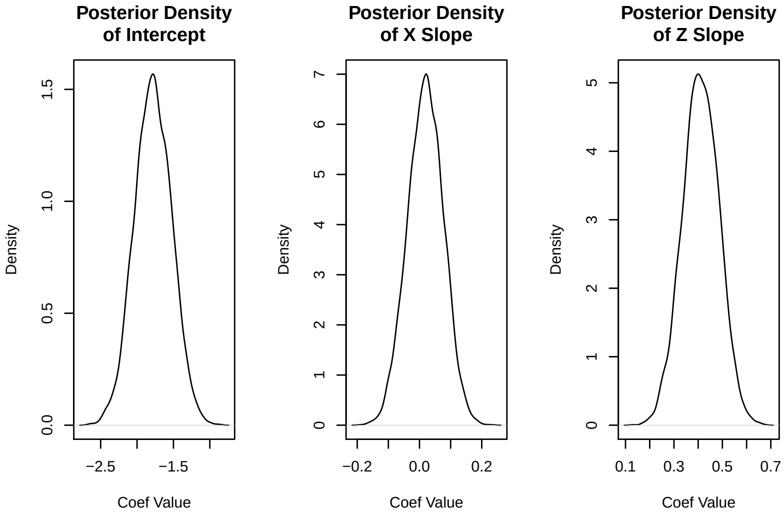
## Bayesian Probit Example II

```
## Sample the posterior imputation model parameters:
coefSams ← matrix(NA, nReps, length(coefs))
for(rp in 1 : nReps) {
    ## Sample the latent response indicators:
    yStar[ySuccess] ← rtnorm(sum(ySuccess),
                             mean = yHats[ySuccess],
                             sd = 1.0,
                             lower = 0)
    yStar[!ySuccess] ← rtnorm(sum(!ySuccess),
                              mean = yHats[!ySuccess],
                              sd = 1.0,
                              upper = 0)

    ## Sample the regression coefficients:
    coefMean ← .lm.fit(y = yStar, x = predMat)$coef
    coefs ← matrix(rmvnorm(1, coefMean, coefVar))
    yHats ← predMat %*% coefs

    ## Store the coefficient samples for later:
    coefSams[rp, ] ← coefs
}
```

## Posterior Parameter Distributions



## Create Imputations

```
## Create the imputations:
sampleIndex ← sample(c(1 : nReps), nImps)
predMat ← as.matrix(cbind(1, yMis[ , c("x", "z")]))
nObs ← nrow(yMis)
impList ← list()
counter ← 0
for(m in sampleIndex) {
    counter ← counter + 1

    ## 1) Compute the predicted values on the z-scale
    ## 2) Convert these values to predicted probabilities
    ## 3) Check if each predictive probability exceeds
    ##    a standard uniform variate.
    ## 4) If 'yes' to (3) impute a 1, else impute a 0
    yHats ← predMat %*% coefSams[m, ]
    imps ← as.numeric(pnorm(yHats) ≥ runif(nObs))

    ## Fill the missing data with the imputations:
    impList[[counter]] ← missData
    impList[[counter]][is.na(missData[ , "y"]), "y"] ←
        imps
}
```

## Same Idea with **mice**

```r
## Try the analogous process in mice():
missData2 ← as.data.frame(missData)
missData2$y ← factor(missData2$y)
## Create the imputations:
miceOut ← mice(missData2,
                m = nImps,
                method = "logreg",
                maxit = 10,
                printFlag = FALSE)
## Fill the missing data with the imputations:
impList2 ← list()
for(m in 1 : nImps) impList2[[m]] ← complete(miceOut, m)
```

## Fit the analysis models I

```
## Fit the analysis model to each imputation:
fitList ← lapply(impList,
                 FUN = function(impData) {
                     glm(y ~ x + z,
                         data = as.data.frame(impData),
                         family = binomial("probit"))
                 })
## Fit the analysis models to mice's imputed data:
fitList2 ← lapply(impList2,
                  FUN = function(impData) {
                      glm(y ~ x + z,
                          data = as.data.frame(impData),
                          family = binomial("probit"))
                  })
## Fit the complete data models:
glmOut ← glm(y ~ x + z,
              data = as.data.frame(simData),
              family = binomial("probit"))
```

## Fit the analysis models II

```
## Compare the results:
coef(MIcombine(fitList)) # By Hand
```

```
(Intercept)           x           z
-1.75928476  0.01737044  0.40571166
```

```
coef(MIcombine(fitList2)) # With mice()
```

```
(Intercept)           x           z
-1.74696283  0.01144854  0.40399215
```

```
coef(glmOut) # Using complete data
```

```
(Intercept)           x           z
-1.72581523  0.01980291  0.39418676
```

## General MI Framework

Almost any proper implementation of MI amounts to out-of-sample prediction using a Bayesian generalized linear model. So, specifying an MI model will entail three steps:

1. Choose a random component
    - The distribution assumed for the missing data.
2. Choose a systematic component
    - The predictors of the imputation model
3. Choose a link function
    - The transformation linking the systematic component to the mean of the random component
    - Often chosen hand-in-hand with the random component

Each step in this process carries its own challenges.

## The End

Okay, let's talk term projects!