Multivariate Multiple Imputation

Utrecht University Winter School: Missing Data in R

Utrecht University

Kyle M. Lang

Department of Methodology & Statistics
Utrecht University

2022-02-03

Outline

Flavors of MI

Fully Conditional Specification

Joint Modeling

Joint Modeling vs. Fully Conditional Specification

When imputing with *Joint Modeling* (JM) approaches, the missing data are replaced by samples from the joint posterior predictive distribution.

- To impute X, Y, and Z, we draw:

$$X, Y, Z \sim P(X, Y, Z|\theta)$$

With *Fully Conditional Specification* (FCS), the missing data are replaced with samples from the conditional posterior predictive distribution of each incomplete variable.

- To impute X, Y, and Z, we draw:

$$X \sim P(X|Y, Z, \theta_X)$$
$$Y \sim P(Y|X, Z, \theta_Y)$$
$$Z \sim P(Z|Y, X, \theta_Z)$$

Joint Modeling: Strengths

When correctly implemented, JM approaches are guaranteed to produce *Bayesianly proper* imputations.

- A sufficient condition for *properness* is that the imputations are randomly sampled from the correctly specified joint posterior predictive distribution of the missing data.
  - This is the defining characteristic of JM methods.

When using the correct distribution, imputations produced by JM methods will be the best possible imputations.

- Unbiased parameter estimates
- Well-calibrated sampling variability

Joint Modeling: Weaknesses

JM approaches don't scale well.

- The computational burden increases with the number of incomplete variables.

JM approaches are only applicable when the joint distribution of all incomplete variables follows a known form.

- Mixes of continuous and categorical variables are difficult to accommodate.

Software Implementations

Fully Conditional Specification: Strengths

FCS scales much better than JM.

- FCS only samples from a series of univariate distributions, not large joint distributions.

FCS approaches can create imputations for variables that don't have a sensible joint distribution.

- FCS can easily treat mixes of continuous and categorical variables.

Fully Conditional Specification: Weaknesses

FCS will usually be slower than JM.

- Each variable gets its own fully parameterized distribution, even if that granularity is unnecessary.

When the incomplete variables don't have a known joint distribution, FCS doesn't have theoretical support.

- There is, however, a large degree of empirical support for the tenability of the FCS approach.
- In practice, we usually choose FCS since real data rarely arise from a known joint distribution.

Software Implementations

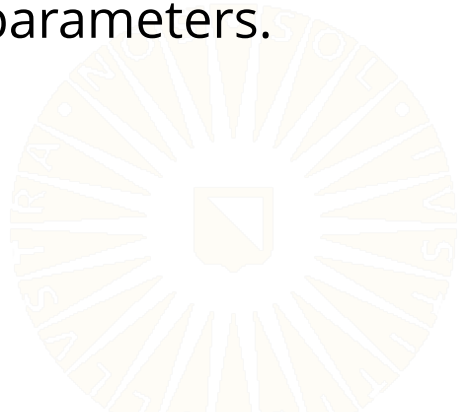
Aside: Gibbs Sampling

Up to this point, most of the models we've explored could be approximated by sampling directly from their posterior distributions.

- This won't be true with arbitrary, multivariate missing data.

To make inference regarding a multivariate distribution with multiple, interrelated, unknown parameters, we can use *Gibbs sampling*.

- Sample from the conditional distribution of each parameter, conditioning on the current best guesses of all other parameters.



Aside: Gibbs Sampling

Suppose the following:

1. I want to make some inference about the tri-variate mean of $X, Y, Z = \mu_X, \mu_Y, \mu_Z \sim P(\mu|\theta)$
2. $P(\mu|\theta)$ is super hairy and difficult to sample
3. I can easily sample from the conditional distributions: $P(\mu_X|\hat{\mu}_Y, \hat{\mu}_Z, \theta)$, $P(\mu_Y|\hat{\mu}_X, \hat{\mu}_Z, \theta)$, and $P(\mu_Z|\hat{\mu}_X, \hat{\mu}_Y, \theta)$.

Then, I can approximate the full joint distribution $P(\mu|\theta)$ by sequentially sampling from $P(\mu_X|\hat{\mu}_Y, \hat{\mu}_Z, \theta)$, $P(\mu_Y|\hat{\mu}_X, \hat{\mu}_Z, \theta)$, and $P(\mu_Z|\hat{\mu}_X, \hat{\mu}_Y, \theta)$.

Aside: Gibbs Sampling

Starting with initial guesses of μ_Y, $\hat{\mu}_Y^{(0)}$, and μ_Z, $\hat{\mu}_Z^{(0)}$, and assuming θ is known, Gibbs sampling proceeds as follows:

$$\hat{\mu}_X^{(1)} \sim P(\mu_X | \hat{\mu}_Y^{(0)}, \hat{\mu}_Z^{(0)}, \theta)$$
$$\hat{\mu}_Y^{(1)} \sim P(\mu_Y | \hat{\mu}_X^{(1)}, \hat{\mu}_Z^{(0)}, \theta)$$
$$\hat{\mu}_Z^{(1)} \sim P(\mu_Z | \hat{\mu}_Y^{(1)}, \hat{\mu}_X^{(1)}, \theta)$$

$$\hat{\mu}_X^{(2)} \sim P(\mu_X | \hat{\mu}_Y^{(1)}, \hat{\mu}_Z^{(1)}, \theta)$$
$$\hat{\mu}_Y^{(2)} \sim P(\mu_Y | \hat{\mu}_X^{(2)}, \hat{\mu}_Z^{(1)}, \theta)$$
$$\hat{\mu}_Z^{(2)} \sim P(\mu_Z | \hat{\mu}_Y^{(2)}, \hat{\mu}_X^{(2)}, \theta)$$
$$\vdots$$

Why do we care?

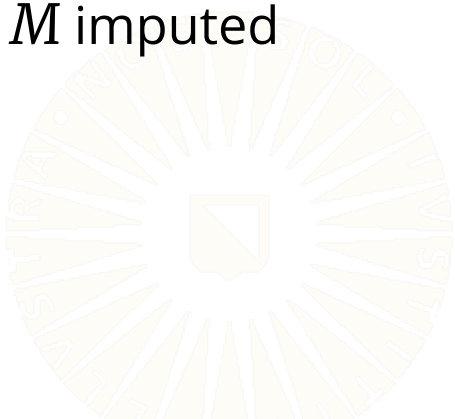
Multivariate MI employs the same logic as Gibbs sampling.

- The imputations are created by conditioning on the current estimates of the imputation model parameters.
- The imputation model parameters are updated by conditioning on the most recent imputations.
- With FCS, each variable is imputed by conditioning on the most recent imputations of all other variables.

Fully Conditional Specification

Procedure: Fully Conditional Specification

1. Fill the missing data with reasonable guesses.
2. For each incomplete variable, do a single iteration of univariate Bayesian MI (e.g., as seen in the last set of slides).
    - After each variable on the data set is so treated, we've completed one iteration.
3. Repeat Step 2 many times.
4. After the imputation model parameters stabilize, save M imputed data sets.



Example: Fully Conditional Specification

```
## Simulate some data:
simData <-
    simCovData(nObs = 1000, sigma = 0.25, nVars = 4)

head(simData, 10)
```

	x1	x2	x3	x4
1	0.06313632	-1.4057704	-0.01709217	1.47929405
2	-1.31592547	1.2970920	-0.83500777	-0.44528158
3	-0.30997023	0.9782580	0.02731853	0.35507390
4	0.06927787	0.1836032	0.68794409	0.08049987
5	-0.99354894	-0.3038956	0.80918329	-1.72143555
6	0.36828016	-0.9423245	1.05155348	-0.11078496
7	1.33333163	2.3089780	1.47203000	0.85877495
8	-0.02759718	0.1714383	0.18927909	0.28627771
9	2.37929433	2.5080935	2.18344726	1.56980951
10	0.31841502	0.7886025	0.73658136	0.39445970

Example: Fully Conditional Specification

```
## Impose missing:
targets  <- paste0("x", 1:3)
missData <- imposeMissData(data    = simData,
                           targets = targets,
                           preds   = "x4",
                           pm      = 0.3,
                           types = c("low", "center", "high")
                           )
```

Example: Fully Conditional Specification

```
head(missData, 10)
```

```
           x1         x2          x3          x4
1          NA -1.4057704 -0.01709217  1.47929405
2  -1.3159255  1.2970920          NA -0.44528158
3  -0.3099702         NA  0.02731853  0.35507390
4          NA  0.1836032  0.68794409  0.08049987
5  -0.9935489 -0.3038956  0.80918329 -1.72143555
6   0.3682802 -0.9423245          NA -0.11078496
7   1.3333316  2.3089780          NA  0.85877495
8          NA         NA  0.18927909  0.28627771
9          NA         NA          NA  1.56980951
10         NA  0.7886025  0.73658136  0.39445970
```

Example: Fully Conditional Specification

```
## Define iteration numbers:
nImps <- 100
nBurn <- 500
nSams <- nBurn + nImps

## Summarize missingness:
rMat <- !is.na(missData)
nObs <- colSums(rMat)
nMis <- colSums(!rMat)

## Fill the missingness with initial (bad) guesses:
mean0  <- colMeans(missData, na.rm = TRUE)
sigma0 <- cov(missData, use = "pairwise")
draws0 <- rmvnorm(nrow(missData), mean0, sigma0)

impData         <- missData
impData[!rMat] <- draws0[!rMat]
```

Example: Fully Conditional Specification

Define an elementary imputation function:

```
eif <- function(data, rVec, v) {
    ## Get the expected betas:
    fit  <- lm(paste(v, "~ ."), data = data[rVec, ])
    beta <- coef(fit)

    ## Sample sigma:
    sigScale <- (1 / fit$df) * crossprod(resid(fit))
    sigmaSam <- rinvchisq(1, df = fit$df, scale = sigScale)

    ## Sample beta:
    betaVar <- sigmaSam * solve(crossprod(qr.X(fit$qr)))
    betaSam <- rmvnorm(1, mean = beta, sigma = betaVar)

    ## Return a randomly sampled imputation:
    matrix(cbind(1, data[!rVec, ])) %*% betaSam +
        rnorm(sum(!rVec), 0, sqrt(sigmaSam))
}
```

Example: Fully Conditional Specification

Apply the elementary imputation function to each incomplete variable:

```
## Iterate through the FCS algorithm:
impList <- list()
for(s in 1 : nSams) {
    for(v in targets) {
        rVec               <- rMat[ , v]
        impData[!rVec, v] <- eif(impData, rVec, v)
    }

    ## If the chains are burnt-in, save imputed datasets:
    if(s > nBurn) impList[[s - nBurn]] <- impData
}
```

```
Error in matrix(cbind(1, data[!rVec, ])) %*% betaSam:  requires
numeric/complex matrix/vector arguments
```

Example: Fully Conditional Specification

Analyze the multiply imputed datasets:

```
## First, our manual version:
fits1 <- lapply(impList,
                function(x) lm(x1 ~ x2 + x3, data = x)
                )
pool1 <- MIcombine(fits1)

Error in variances[[1]]:  subscript out of bounds

## Do the same analysis with mice():
miceOut <- mice(data      = missData,
                m         = 100,
                method    = "norm",
                printFlag = FALSE)
fits2 <- with(miceOut, lm(x1 ~ x2 + x3))
pool2 <- pool(fits2)
```

Example: Fully Conditional Specification

Compare approaches:

```
Error in coef(pool1):  object 'pool1' not found
Error in vcov(pool1):  object 'pool1' not found
Error in eval(expr, envir, enclos):  object 'cf1' not found
Error in eval(expr, envir, enclos):  object 'pool1' not found
Error in pt(t1, df1, lower.tail = FALSE): object 't1' not
found
Error in eval(expr, envir, enclos):  object 'pool1' not found
Error in cbind(cf1, se1, t1, p1, fmi1):  object 'cf1' not
found
Error in colnames(res1) <- c("Est", "SE", "t", "p", "FMI"):
object 'res1' not found
Error in eval(expr, envir, enclos):  object 'res1' not found
Error in res1[flag, "p"] <- "<0.001":  object 'res1' not
found
Error in rownames(res1):  object 'res1' not found
Error in xtable(res1, caption = "Manual Version"):  object
'res1' not
```

Joint Modeling

Aside: Definition of Regression Parameters

So far, we've been using the least-squares estimates of α, β, and σ^2 to parameterize our posterior distributions.

- We can also define the parameters in terms of sufficient statistics.

Given μ and Σ, we can define all of our regression moments as:

$$\begin{aligned}
\beta &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \\
&= \text{Cov}(\mathbf{X})^{-1}\text{Cov}(\mathbf{X}, \mathbf{Y}) \\
\alpha &= \mu_Y - \beta^T\mu_X \\
\Sigma_\varepsilon &= \Sigma_Y - \beta^T\Sigma_X\beta
\end{aligned}$$

These definitions are crucial for JM approaches.

- Within the subset of data define by a given response pattern, the outcome variables will be entirely missing.

Multivariate Bayesian Regression

Previously, we saw examples of univariate Bayesian regression which used the following model:

$$\beta \sim \text{MVN}\left(\hat{\beta}_{ls},\ \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}\right)$$
$$\sigma^2 \sim \text{Inv-}\chi^2\left(N-P, MSE\right)$$

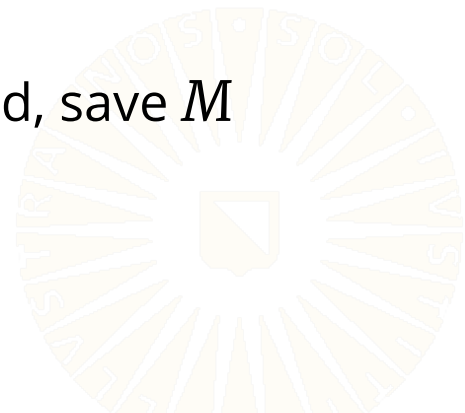
We can directly extend the above to the multivariate case:

$$\Sigma^{(i)} \sim \text{Inv-W}\left(N-1, (N-1)\Sigma^{(i-1)}\right)$$
$$\mu^{(i)} \sim \text{MVN}\left(\mu^{(i-1)}, N^{-1}\Sigma^{(i)}\right)$$

We get α, β, and Σ_ε via the calculations on the preceding slide

Procedure: Joint Modeling

1. Partition the incomplete data by response pattern.
   - Produce S subsets wherein each row shares the same response pattern.
2. Provide initial guesses for μ and Σ.
3. Within each subset, use the current guesses of μ and Σ to generate imputations via multivariate Bayesian regression.
4. Use the filled-in data matrix to updated the sufficient statistics.
5. Repeat Steps 3 and 4 many times.
6. After the imputation model parameters have stabilized, save M imputed data sets produced in Step 3.



Example: Joint Modeling

```
iStep <- function(data, pats, ind, p0, pars) {
    ## Loop over non-trivial response patterns:
    for(i in c(1 : nrow(pats))[-p0]) {
        ## Define the current response pattern:
        p1 <- pats[i, ]

        ## Subset the data:
        dat1 <- data[ind == i, ]

        ## Replace missing data with imputations:
        data[ind == i, !p1] <- getImps(data  = dat1,
                                       mu    = pars$mu,
                                       sigma = pars$sigma,
                                       p1    = p1)
    }

    ## Return the imputed data:
    data
}
```

Example: Joint Modeling

```
getImps <- function(data, mu, sigma, p1) {
    ## Partition the parameter matrices:
    mY  <- matrix(mu[!p1])
    mX  <- matrix(mu[p1])
    sY  <- sigma[!p1, !p1]
    sX  <- sigma[p1, p1]
    cXY <- sigma[p1, !p1]

    ## Compute the imputation model parameters:
    beta  <- solve(sX) %*% cXY
    alpha <- mY - t(beta) %*% mX
    sE    <- sY - t(beta) %*% sX %*% beta

    ## Pull out predictors:
    X <- as.matrix(data[ , p1])

    ## Generate and return the imputations:
    n <- nrow(X)
    matrix(1, n) %*% t(alpha) + X %*% beta + rmvnorm(n, sigma = sE)
}
```

Example: Joint Modeling

```
pStep <- function(data) {
    ## Update the complete-data sufficient statistics:
    n <- nrow(data)
    m <- colMeans(data)
    s <- (n - 1) * cov(data)

    ## Sample sigma and mu:
    sigma <- riwish((n - 1), s)
    mu    <- rmvnorm(1, m, (sigma / n))

    ## Return the updated parameters:
    list(mu = mu, sigma = sigma)
}
```

Example: Joint Modeling

Now that we've defined the necessary functions, do the imputation:

```
## Some preliminaries:
impData <- missData
nIter   <- 50
nImps   <- 100

## Summarize response patterns:
rMat <- !is.na(impData)
pats <- uniquecombs(rMat)

Error in uniquecombs(rMat):  could not find function "uniquecombs"

ind  <- attr(pats, "index")

Error in eval(expr, envir, enclos):  object 'pats' not found

p0   <- which(apply(pats, 1, all))

Error in apply(pats, 1, all):  object 'pats' not found

## Get starting values for the parameters:
pars <- list(mu    = colMeans(impData, na.rm = TRUE),
             sigma = cov(impData, use = "pairwise")
```

Example: Joint Modeling

```
## Iterate over I- and P-Steps to generate imputations:
impList3 <- list()
for(m in 1 : nImps) {
    for(rp in 1 : nIter) {
        impData <- iStep(data = impData,
                         pats = pats,
                         ind  = ind,
                         p0   = p0,
                         pars = pars)
        pars <- pStep(impData)

        if(rp == nIter) impList3[[m]] <- impData
    }
}
```

```
Error in iStep(data = impData, pats = pats, ind = ind, p0 = p0, pars =
pars):  object 'pats' not found
```

Example: Joint Modeling

Do the same type of imputation with `norm`:

```
missData <- as.matrix(missData)

meta   <- prelim.norm(missData)

Error in prelim.norm(missData):  could not find function "prelim.norm"

theta0 <- em.norm(meta, showits = FALSE)

Error in em.norm(meta, showits = FALSE): could not find function "em.norm"

rngseed(235711)

Error in rngseed(235711):  could not find function "rngseed"

impList4 <- list()
for(m in 1 : nImps) {
    theta1 <- da.norm(s     = meta,
                      start = theta0,
                      steps = nIter)
    impList4[[m]] <- imp.norm(s     = meta,
                              theta = theta1,
                              x     = missData)
}
```

Example: Joint Modeling

Analyze the multiply imputed data:

```
## Manual implementation:
fits3 <- lapply(impList3,
                function(x) lm(x1 ~ x2 + x3, data = x)
                )
pool3 <- MIcombine(fits3)

Error in variances[[1]]:  subscript out of bounds

## Imputation using norm():
fits4 <- lapply(impList4,
                function(x) lm(x1 ~ x2 + x3,
                               data = as.data.frame(x)
                               )
                )
pool4 <- MIcombine(fits4)

Error in variances[[1]]:  subscript out of bounds
```

Example: Joint Modeling

Compare approaches:

```
Error in coef(pool3):  object 'pool3' not found
Error in vcov(pool3):  object 'pool3' not found
Error in eval(expr, envir, enclos):  object 'cf3' not found
Error in eval(expr, envir, enclos):  object 'pool3' not found
Error in pt(t3, df3, lower.tail = FALSE): object 't3' not
found
Error in eval(expr, envir, enclos):  object 'pool3' not found
Error in cbind(cf3, se3, t3, p3, fmi3):  object 'cf3' not
found
Error in colnames(res3) <- c("Est", "SE", "t", "p", "FMI"):
object 'res3' not found
Error in eval(expr, envir, enclos):  object 'res3' not found
Error in res3[flag, "p"] <- "<0.001":  object 'res3' not
found
Error in coef(pool4):  object 'pool4' not found
Error in vcov(pool4):  object 'pool4' not found
```