CatContModelDA_Handbook

Xin Ye, Kyle Hardman

6/11/2022

Contents

1 Introduction

This is a handbook for how to conduct data analysis by CatContModel developed by Dr. Kyle Hardman. It includes instructions for analyzing [1,180] data and how to choose models for your dataset. I will use the [1,180] dataset collected by XinYe 2022 as an example.

2 How to handle [1,180] data?

This kind of data is generated from a bar memory task. When we use the orientation of the bar as the stimuli, the response value will range from 1 to 180 degrees. However, the default circular variable range for CatContModel is [1,360], which means we cannot use the CatContModel package directly under that situation.

Luckily, relative method and modification to codes has provided by Dr. Hardman:

The thoughts of the method: We could expand the range of the data to 360 degrees by multiplying all of the study and response values by 2. This solution respects the data but will require additional post-processing of estimated parameter values for purposes of describing and plotting the parameters. Since the data have been scaled up to [0,360], the parameters will need to be scaled back down as though the data had been in [0,180]. In particular, we will need to divide by 2 all of the standard deviation parameters (catSd, catSelectivity, and contSD) and the category location parameters (catMu).

With that solution, there are complexities with hypothesis testing, because the priors on the standard deviation parameters will be wrong after dividing the posteriors by 2. (The package does not have any hypothesis tests on catMu, so there is no concern there.) Hypothesis tests can still be done with the original, undivided posteriors. For descriptive statistics, like posterior means and plots, use the divided posteriors.

To be concluded, the standard deviation parameters (contSD, catSD, and catSelectivity) and catMu will be double their true values. Therefore, for plots and other descriptive statistics, these parameters must be scaled by dividing by 2. For hypothesis tests, however, the original, undivided parameter values must be used.

Relevant code are provided below:

2.1 Preprocessing data

At the stage of preprocessing data, we need to modify the data by multiplying by 2, changing the range from [0,180] to [0,360].

```
data2x = data # copy the data before modification

# Modify the data by multiplying by 2
data2x$study = data2x$study * 2
data2x$response = data2x$response * 2

# Check the ranges and plot the data
range(data2x$study)
range(data2x$response)
plot(data2x$study, data2x$response)

# Set up a basic configuration and run the model
config2x = list(modelVariant="withinItem",
                iterations=11000,
                maxCategories=10,
                responseRange = c(1,360))
```

2.2 Descriptive Statistics

After finishing mTuning and getting the output, we need to draw plots and other descriptive statistics.

```
W_results = readRDS("WithinItem_results_bar180.RDS")

#remove burn in
W_results = removeBurnIn(W_results, burnIn=1000)

ssw=data.frame()
for (i in  1:11){
  ssw[i,1] = mean(getParameterPosterior(W_results, param="pMem", pnum=i, cond='pos'))
  ssw[i,2] = mean(getParameterPosterior(W_results, param="pMem", pnum=i, cond='neu'))
  ssw[i,3] = mean(getParameterPosterior(W_results, param="pMem", pnum=i, cond='neg'))
  ssw[i,4] = mean(getParameterPosterior(W_results, param="contSD", pnum=i, cond='pos'))
  ssw[i,5] = mean(getParameterPosterior(W_results, param="contSD", pnum=i, cond='neu'))
  ssw[i,6] = mean(getParameterPosterior(W_results, param="contSD", pnum=i, cond='neg'))
  ssw[i,7] = mean(getParameterPosterior(W_results, param="pContWithin", pnum=i, cond='pos'))
  ssw[i,8] = mean(getParameterPosterior(W_results, param="pContWithin", pnum=i, cond='neu'))
  ssw[i,9] = mean(getParameterPosterior(W_results, param="pContWithin", pnum=i, cond='neg'))
}
```

Also, we need to add two new functions for future analysis. You could directly add it in your code chunk.

```
# [0,360] degrees
clampAngle_R = function(xs, pm180=FALSE) {

  xs = xs %% 360

  if (pm180) {
    xs[ xs > 180 ] = xs[ xs > 180 ] - 360
  }
  xs
}
```

```
# signed distance in degrees
circDist_R = function(xs, ys) {
  clampAngle_R(ys - xs, pm180=TRUE)
}
```

2.3 Descriptive Analysis

As we have mentioned before, to describe the standard deviation and catMu parameter posteriors with respect to the original half-circle data space, those parameter posteriors must be divided by 2.

The function(halfCircle_convertParameters) below converts posterior parameter chains to/from half circle space. The standard deviation parameters (contSD, catSD, and catSelectivity) and catMu posteriors will be multiplied by multiple.

```
halfCircle_convertParameters = function(res, multiple) {

  fullParamNames = names(res$posteriors)

  # Select standard deviation parameters from all parameters
  sdParamNames = getSdParams(res)
  fullSdParam = NULL
  for (sdp in sdParamNames) {
    matchSd = grepl(sdp, fullParamNames, fixed=TRUE)
    fullSdParam = c(fullSdParam, fullParamNames[ matchSd ])
  }

  # Modify standard deviation parameters
  for (sdp in fullSdParam) {

    post = res$posteriors[[ sdp ]]

    if (grepl(".var", sdp, fixed=TRUE)) {
      # Variance parameters are in squared units, so sqrt, divide, then square
      post = sqrt(post)
      post = post * multiple
      post = post^2
      # Adjusting variance parameters like this may not be totally correct, but approximate.
    } else {
      # Participant and mean (.mu) parameters divide by 2
      post = post * multiple
    }

    res$posteriors[[ sdp ]] = post
  }

  # Modify catMu
  fullCatMu = fullParamNames[ grepl("catMu", fullParamNames, fixed=TRUE) ]
  for (cm in fullCatMu) {

    post = res$posteriors[[ cm ]]

    # The catMu parameters are free to wander outside of [0,360].
    # Bring them back before multiplying
```

```
    #post = CatContModel::clampAngle(post, pm180 = FALSE, degrees = TRUE)
    post = clampAngle_R(post, pm180 = FALSE)

    post = post * multiple

    res$posteriors[[ cm ]] = post
  }

  res
}
```

After that, we could start to plot all parameters.

```
# res2x was estimated with data that were multiplied by 2 to get to the full circle space.
# Get back to the half circle space by dividing by 2 (i.e. multiply by 1/2)
res2x_conv = halfCircle_convertParameters(W_results, multiple=1/2)

# Means and credible intervals are descriptive and use converted parameters
posteriorMeansAndCredibleIntervals(res2x_conv)

# Plots are also descriptive and use converted parameters.
# The catMu plot has some problems, but is still readable
plotParameterSummary(res2x_conv)

# You can plot individual panels of the overall plot with plotParameter
plotParameter(res2x_conv, param="contSD")
plotParameter(res2x_conv, param="catSelectivity")
plotParameter(res2x_conv, param="catSD")
plotParameter(res2x_conv, param="pMem")
plotParameter(res2x_conv, param="pContWithin")
plotParameter(res2x_conv, param="pCatGuess")

# This function is a possible way to plot catMu for half circle data
plotCatMu_halfCircle = function(res, precision = 1, pnums = res$pnums) {

  post = convertPosteriorsToMatrices(res, param = c("catMu", "catActive"))

  catMu = post$catMu[ pnums, , ]
  catActive = post$catActive[ pnums, , ]

  plotData = CatContModel:::catMu_getPlotData(catMu, catActive, precision = precision,
                                              dataType = res$config$dataType,
                                              responseRange = res$config$responseRange)

  # Ignore all points past 180
  plotData = plotData[ plotData$x <= 180, ]

  # The last point is the first point, so copy the value from the first point.
  plotData$y[ plotData$x == 180 ] = plotData$y[1]

  CatContModel:::catMu_plotSetup(plotData, res$config$dataType)
  CatContModel:::catMu_plotData(plotData$x, plotData$y, col = plotData$color, type = "polygon")
```

```
  invisible(plotData)
}

plotCatMu_halfCircle(res2x_conv)
```

2.4 Statistical analysis

At last, we need to conduct relevant statistical analysis. Codes below provided how to conduct statistical analysis for [1,180]data.

```
# Although converted parameters are used for describing the results, other kinds of
# analysis will not work correctly if you use the converted parameter values.
# Here are some major examples of where you should use the original parameter values.

# Hypothesis tests depend on priors, so use unconverted parameters
# Testing Main Effects and Interactions
mei = testMainEffectsAndInteractions(W_results,doPairwise = TRUE)
# Examine bayes factors in favor of the effect.
mei[ mei$bfType == "10", ]
# Select tests with reasonably strong BFs (either for or against an effect).
mei[ mei$bf > 3, ]


ce = testConditionEffects(W_results)
ce[ ce$bfType == "10", ]


# Posterior predictive (sampling data based on parameters) uses unconverted parameters
posteriorPredictivePlot(W_results)

# WAIC also uses unconverted parameters
calculateWAIC(W_results)

# There are many functions in this package that summarize results.
# When using each, think carefully about whether the function wants
# the original real posteriors or converted posteriors.
# Anything that uses priors must use original posteriors.
```

3 How to choose a model?

Another important topic we need to know is how to choose an appropriate model for your dataset in CatContModel. There are three models which could be chosen in CatContModel. It is important to fit your data into a better model to prevent from getting biased results. Notice: this section can be also applied to all ranges of data not just for [1,180] data.

3.1 Step1: Get an idea of the pCatgory

First, we could check model parameters to see whether there is categorical responses. The tests we could use to determine if there is a non-negligible amount of categorical responding are in the function testCategorical-Responding. It tests if memory is almost totally continuous by testing if pContBetween or pContWithin are

basically 1 (depending on the model). It tests for the presence of categorical guessing by testing if pCatGuess is basically 0. If these tests shows that participants are not using categories(the value of pCatGuess is quite small), then the ZL model is probably right.

Actually, regardless of whether there is categorical responding visible in the data plots, we can fit data to any of the models. After fitting the models, if the estimated amount of categorical responding is negligible, then the ZL model is a good choice.

3.2 Step2: Parameter convergence

One thing to point out, the first step before choosing a model is to make sure that parameters are convergent in the model results. Convergence is supposed to be checked before any results from a model are used, including things like WAIC.

In Bayesian parameter estimation, parameters should converge to a stable distribution. If a parameter does not converge to a stable distribution, then the distribution of that parameter has not been found and plausible values for that parameter are unknown. In other words, the parameter estimation failed to provide useful estimates of the parameter. If parameters do not converge, you cannot use the parameter estimates because they are unreliable/wrong values.

There are various statistics that can be calculated to test for convergence, but plotting the parameters and looking at the plots can tell you all you need to know. More details will be illustrated below.

As for codes of plotting convergence plots, please check the code "Convergence Plots.R" below. With the code, you are able to make a big pdf with all the plots.

```
# paramGroup like pMem, contSD, catMu, etc.
getOrderedMatchingParameterNames = function(res, paramGroup) {

  paramNames = names(res$posteriors)

  # Find the parameters and put them in order
  matchingParam = grepl(paste0("^", paramGroup), paramNames)
  matchingParam = paramNames[ matchingParam ]

  # put .mu, .var, and _cond first
  muLoc = which(grepl("\\.mu$", matchingParam))
  varLoc = which(grepl("\\.var$", matchingParam))
  condLocs = which(grepl("_cond", matchingParam))
  paramOrder = c(muLoc, varLoc, condLocs)

  allLocs = 1:length(matchingParam)
  participantLocs = allLocs[ !(allLocs %in% paramOrder) ]

  paramOrder = c(paramOrder, participantLocs)

  matchingParam[ paramOrder ]

}

#' Plots of Posterior Parameter Chains
#'
#' Makes plots, preferably to a pdf file, of posterior parameter chains
# for purposes of visual convergence diagnostics.
#' There are other uses for these plots, including model validation
```

```r
# (are the parameters behaving reasonably?).
#'
#' @param res A results object.
#' @param filename The name of a pdf file to plot to. If the file extension is not ".pdf", t
# hat file extension will be appended.
#' @param whichIterations Numeric vector. Which iterations to plot.
convergencePlots = function(res, filename=NULL, whichIterations=NULL) {
  convergencePlots_parallel(list(res), filename=filename, whichIterations=whichIterations)
}



# mega function to do all parameters in one file
convergencePlots_parallel = function(resList, filename=NULL, whichIterations=NULL) {

  titlePagePlot = function(text, x=0.5, y=0.5, cex=3) {
    graphics::par(mfrow=c(1,1))
    graphics::plot(c(0,1), c(0,1), type='n', axes=FALSE, xlab="", ylab="")
    graphics::text(x, y, text, cex=cex)
  }

  usedPnums = resList[[1]]$pnums

  if (!is.null(filename)) {
    if (!grepl("\\.pdf$", filename)) {
      filename = paste0(filename, ".pdf")
    }

    grDevices::pdf(filename, width = 8, height = 4)
  }

  paramNameGroups = getAllParams(resList[[1]], filter=TRUE)

  # Main title page
  origMar = par()$mar
  par(xpd=TRUE, mar=c(0,0,0,0))
  titlePagePlot("Convergence Diagnostic Plots", y=0.9)

  printedParamNameGroups = paramNameGroups[1:min(length(paramNameGroups), 3)]
  if (length(paramNameGroups) > 3) {
    printedParamNameGroups = c(printedParamNameGroups, "etc.")
  }

  labels = c("For all standard parameters, posterior chains and density are plotted. ",
             "   Population mean and variance: param.mu, param.var",
             "   Effects of task condition: param_cond[cond]",
             "   Participant level parameters: param[pnum]",
             paste0("Standard params: ", paste(printedParamNameGroups, collapse=", "))
             )

  if (resList[[1]]$config$modelVariant != "ZL") {
    labels = c(labels, "Category parameters (catMu, catActive) follow with a different format.")
  }
```

```
text(x=0.05, y=seq(0.7, 0.1, length.out=length(labels)), labels=labels, cex=1.2, adj=0)
par(xpd=FALSE, mar=origMar)

# standard params
for (i in 1:length(paramNameGroups)) {

  titlePagePlot(paramNameGroups[i])

  # TODO: Select participants
  matchingParam = getOrderedMatchingParameterNames(resList[[1]], paramNameGroups[i])

  #graphics::par(mfrow=c(1,2))
  for (i in 1:length(matchingParam)) {
    convergencePlotStandard_parallel(resList, matchingParam[i],
                                     type=c("chain", "density"),
                                     whichIterations=whichIterations,
                                     combinedDens=FALSE, panelize=TRUE)
  }

}


if (resList[[1]]$config$modelVariant != "ZL") {

  # category params are done for each results object separately so participants
  # have their chains together

  # title page
  if (length(resList) > 1) {
    titlePagePlot("Category Parameters", y=0.75)
    labels = c("For each participant:",
               "Multiple chains: catMu density, catActive iterations",
               "Each chain: catMu scatter, catActive iterations")
    text(x=0.5, y=c(0.45, 0.3, 0.15), labels=labels, cex=1.5)
  } else {
    titlePagePlot("Category Parameters", y=0.5)
  }

  # For each participant, plot the multi chain results, then each individual chain
  for (i in 1:length(usedPnums)) {

    if (length(resList) > 1) {
      #convergencePlotCategory_parallel(resList, pnums=usedPnums[i], panelize=TRUE)
      par(mfrow=c(1,1))
      convergencePlot_catMuDensity_parallel(resList, pnums=usedPnums[i],
                                            whichIterations=whichIterations)
      convergencePlot_catActive_parallel(resList, pnums=usedPnums[i],
                                         whichIterations=whichIterations)
    }

    for (resInd in 1:length(resList)) {
      #convergencePlotCategory(resList[[resInd]], pnums=usedPnums[i], panelize=TRUE)
```

```
        par(mfrow=c(1,2))
        convergencePlot_catMuScatter(resList[[resInd]], pnum=usedPnums[i],
                                     whichIterations=whichIterations)
        convergencePlot_catActive_parallel(list(resList[[resInd]]),
                                           pnums=usedPnums[i],
                                           whichIterations=whichIterations)

      }
    }
  }

  if (!is.null(filename)) {
    dev.off()
  }

}

convergencePlotStandard = function(res, param, type=c("chain", "density"),
                                   whichIterations=NULL, panelize=TRUE) {
  convergencePlotStandard_parallel(list(res), param=param, type=type,
                                   whichIterations=whichIterations, c
                                   ombinedDens=FALSE, panelize=panelize)
}


# For non-category parameters (all but catMu and catActive)
# type %in% chain, density
convergencePlotStandard_parallel = function(resList, param, type=c("chain", "density"),
                                            whichIterations=NULL,
                                            combinedDens=FALSE,
                                            panelize=TRUE)
  {

  # check param name
  if (!(param %in% names(resList[[1]]$posteriors)) ||
      grepl("catMu", param, fixed=TRUE) ||
      grepl("catActive", param, fixed=TRUE))
  {
    warning(paste0("Invalid parameter name: \"", param, "\""))
    return()
  }

  # shared setup
  mm = matrix(nrow=resList[[1]]$config$iterations, ncol=length(resList))
  for (i in 1:length(resList)) {
    mm[,i] = resList[[i]]$posteriors[[param]]
  }

  whichIterations = CatContModel:::valueIfNull(whichIterations, 1:resList[[1]]$config$iterations)
  whichIterations = whichIterations[ whichIterations %in% 1:nrow(mm) ]
  mm = mm[whichIterations,,drop=FALSE]

  valRange = range(mm)
```

```
col = grDevices::rainbow(ncol(mm))


# Clean type
type = unique(type[ type %in% c("chain", "density") ])
if (panelize && length(type) == 2) {
  graphics::par(mfrow=c(1,2))
}

# chain
if ("chain" %in% type) {
  base::plot(x=range(whichIterations), y=valRange, type='n', xlab="Iteration", ylab=param)
  for (i in 1:ncol(mm)) {
    graphics::lines(x=whichIterations, y=mm[,i], col=col[i])
  }
  graphics::title(paste0("Chain for ", param))
}


# density
paramIsConstant = valRange[1] == valRange[2]
if ("density" %in% type) {
  require(polspline)

  # Don't try to do density for constant parameters
  if (paramIsConstant) {

    densXs = valRange[1] + c(-1, -0.01, 0, 0.01, 1)
    densMat = matrix(0, nrow=length(densXs), ncol=ncol(mm))
    densMat[3,] = 1

  } else {
    densXs = seq(valRange[1], valRange[2], length.out = 100)

    densMat = matrix(nrow=length(densXs), ncol=ncol(mm))
    for (i in 1:ncol(mm)) {
      ls = polspline::logspline(mm[,i])
      densMat[,i] = polspline::dlogspline(densXs, ls)
    }
  }

  base::plot(x=valRange, y=c(0,max(densMat)), type='n', xlab=param, ylab="Density")
  for (i in 1:ncol(mm)) {
    graphics::lines(x=densXs, y=densMat[,i], col=col[i])
  }
  graphics::title(paste0("Density for ", param))

  # Add combined density line
  if (combinedDens && !paramIsConstant) {
    totalDensLS = polspline::logspline(as.vector(mm))
    graphics::lines(x=densXs, y=polspline::dlogspline(densXs, totalDensLS),
                    col="black", lwd=2, lty=2)
  }
```

```
  }
}




# single chain, single participant
convergencePlot_catMuScatter = function(res, pnum, whichIterations=NULL)
  {

  whichIterations = CatContModel:::valueIfNull(whichIterations, 1:res$config$iterations)

  catCol = grDevices::rainbow(res$config$maxCategories)
  alphaHex = gettextf("%X", 51)
  catCol = paste0(catCol, alphaHex)

  pMats = convertPosteriorsToMatrices(res, param=c("catMu", "catActive"))

  pInd = which(pnum == res$pnums)

  cmi = pMats$catMu[pInd,,]
  cai = pMats$catActive[pInd,,]


  ylim = c(0, 360)
  if (res$config$dataType == "linear") {
    ylim = res$config$catMuRange
  }

  base::plot(x=range(whichIterations), y=ylim, axes=FALSE,
             type='n', xlab="Iteration", ylab="Category Location")
  graphics::box()
  graphics::axis(1)
  graphics::axis(2, at=seq(0, 360, 45))
  graphics::title(paste0("Active catMu for pnum:", pnum))

  for (j in 1:dim(cmi)[1]) {
    cmij = cmi[j,whichIterations]
    caij = cai[j,whichIterations]

    #cmij = CatContModel:::clampAngle(cmij, pm180 = FALSE, degrees = TRUE)
    cmij = clampAngle_R(cmij, pm180 = FALSE)

    cmij[ caij == 0 ] = -100 # This is such a gross hack

    graphics::points(x=whichIterations, y=cmij, pch=18, col=catCol[j]) # pch=20
  }

}


# All participants in one density plot.
# Only active categories
```

```r
convergencePlot_catMuDensity_parallel = function(resList, pnums=NULL,
                                                 catMuPrecision=2, whichIterations=NULL) {
  nRes = length(resList)
  pnums = CatContModel:::valueIfNull(pnums, resList[[1]]$pnums)
  chainColors = grDevices::rainbow(nRes)

  whichIterations = CatContModel:::valueIfNull(whichIterations, 1:resList[[1]]$config$iterations)

  allPlotData = NULL


  for (i in 1:nRes) {
    post = convertPosteriorsToMatrices(resList[[i]], param = c("catMu", "catActive"))


    includedPnumInd = which(dimnames(post$catMu)[[1]] %in% pnums)
    post$catMu = post$catMu[includedPnumInd,,whichIterations]

    includedPnumInd = which(dimnames(post$catActive)[[1]] %in% pnums)
    post$catActive = post$catActive[includedPnumInd,,whichIterations]


    plotData = CatContModel:::catMu_getPlotData(post$catMu, post$catActive,
                                precision = catMuPrecision,
                                dataType = resList[[i]]$config$dataType,
                                responseRange = resList[[i]]$config$responseRange,
                                colorGeneratingFunction = resList[[i]]$config$colorGeneratingFunction)

    plotData$chain = i
    allPlotData = rbind(allPlotData, plotData)

  }

  catMu_plotSetup(allPlotData, resList[[1]]$config$dataType) #, ylim_1 = colorBarYlim[2])

  if (length(pnums) > 1) {
    graphics::title("Active catMu for multiple pnums")
  } else {
    graphics::title(paste0("Active catMu for pnum:", pnums))
  }

  for (i in 1:nRes) {
    plotData = allPlotData[ allPlotData$chain == i, ]
    catMu_plotData(plotData$x, plotData$y, col = chainColors[i], type = "line")
  }

  invisible(allPlotData)
}


# All participants in one chain plot
# plots lines, no density
# type = c("line", "point")
```

```r
convergencePlot_catActive_parallel = function(resList, pnums=NULL, type="line", whichIterations=NULL) {

  nRes = length(resList)
  #nIterations = resList[[1]]$config$iterations
  pnums = CatContModel:::valueIfNull(pnums, resList[[1]]$pnums)
  chainColors = grDevices::rainbow(nRes)

  whichIterations = CatContModel:::valueIfNull(whichIterations, 1:resList[[1]]$config$iterations)

  # per chain, count of active categories averaged across participants
  catActivePlotData = matrix(0, nrow=length(whichIterations), ncol=nRes)

  for (i in 1:nRes) {
    post = convertPosteriorsToMatrices(resList[[i]], param = "catActive")

    includedPnumInd = which(dimnames(post$catActive)[[1]] %in% pnums)
    post$catActive = post$catActive[includedPnumInd,,whichIterations]


    if (length(pnums) > 1) {
      # sum for each participant
      caSum = apply(post$catActive, c(1,3), sum)
      # mean of participants
      catActivePlotData[,i] = apply(caSum, 2, mean)
    } else {
      # sum for the single participant
      catActivePlotData[,i] = apply(post$catActive, 2, sum)
    }

  }

  yRange = range(catActivePlotData)

  base::plot(range(whichIterations), yRange, type='n', xlab="Iteration",
             ylab="Number of active categories")

  if (length(pnums) > 1) {
    graphics::title("Mean of sum of catActive for multiple pnums")
  } else {
    graphics::title(paste0("Sum of catActive for pnum:", pnums))
  }

  for (i in 1:nRes) {
    if (type == "line") {
      graphics::lines(x=whichIterations, y=catActivePlotData[,i], col=chainColors[i], lwd=1)
    } else if (type == "point") {
      graphics::points(x=whichIterations, y=catActivePlotData[,i], col=chainColors[i], pch=18)
    }
  }

  invisible(catActivePlotData)
}
```

```
convergencePlots(W_results, filename="plots.pdf", whichIterations=NULL)
```

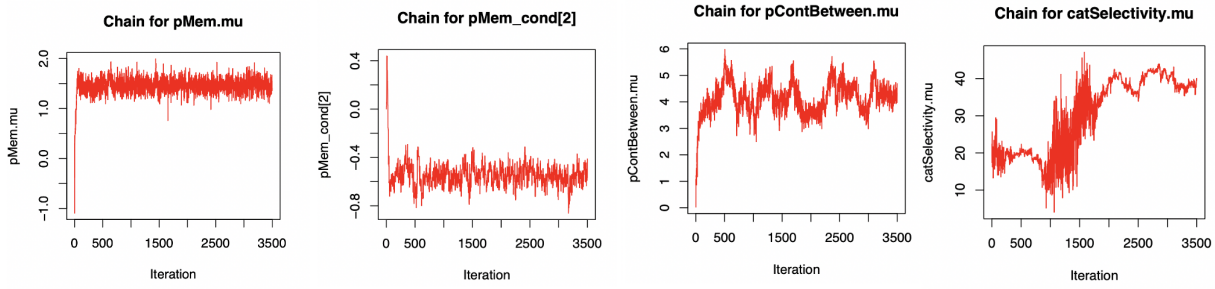
Notice: the "whichIterations" parameter need to be set to NULL.

3.2 How to make sure that the parameter is convergent?

Standard Parameters

When looking at the convergence plots, there are some characteristics to look for depending on parameter type. For the standard model parameters (e.g. pMem), focus on the grand mean parameters (.mu) and condition effects (_cond).

As for a good convergence example, the value would quickly moved to a distribution centered around a constant value. The shape (width) and location of the distribution is consistent over thousands of iterations, so the parameter has converged. On the contrary, examples of bad or no convergence means the chain never reaches a nice stable distribution.



Some individual participant parameters may not converge nearly as well as the means and condition effects. This does not mean that you need to throw out those participants. (I wouldn't throw out a participant based on convergence unless many of their parameters look extremely bad.)

Category Parameters

For the category parameters (catMu and catActive), the convergence plots look different and are done for one participant at a time.

For catActive, the number of active categories should converge to a reasonably stable distribution. For some participants, the number of active categories is almost constant and looks like a flat line with a few spikes up or down.

For catMu, the scatterplot shows only active categories (catActive = 1). You should see straight lines going across the plot indicating that the model has found a category at that location. (You can ignore the color of the dots, which represents category index, because all catMu parameters are exchangeable.) The number of lines going across the catMu plot should be the same as the sum of catActive.

Examples of good category parameter convergence: As shown in figure1 "Good Example1", there will be clear straight lines across catMu and the number of lines matches the sum of catActive.For some of participants, the number of categories were high and/or categories were close together so the plots are messier. Still, the sum of catActive is not hugely variable and the catMu are in reasonably stable places.

Examples of bad category parameter convergence: As shown in figure2 "Bad Example2", for all participants, the catMu plot is pure noise. There are no distinctive locations that are identified as consistently having categories. Instead, categories are tried everywhere and the sum of catActive is hugely variable.

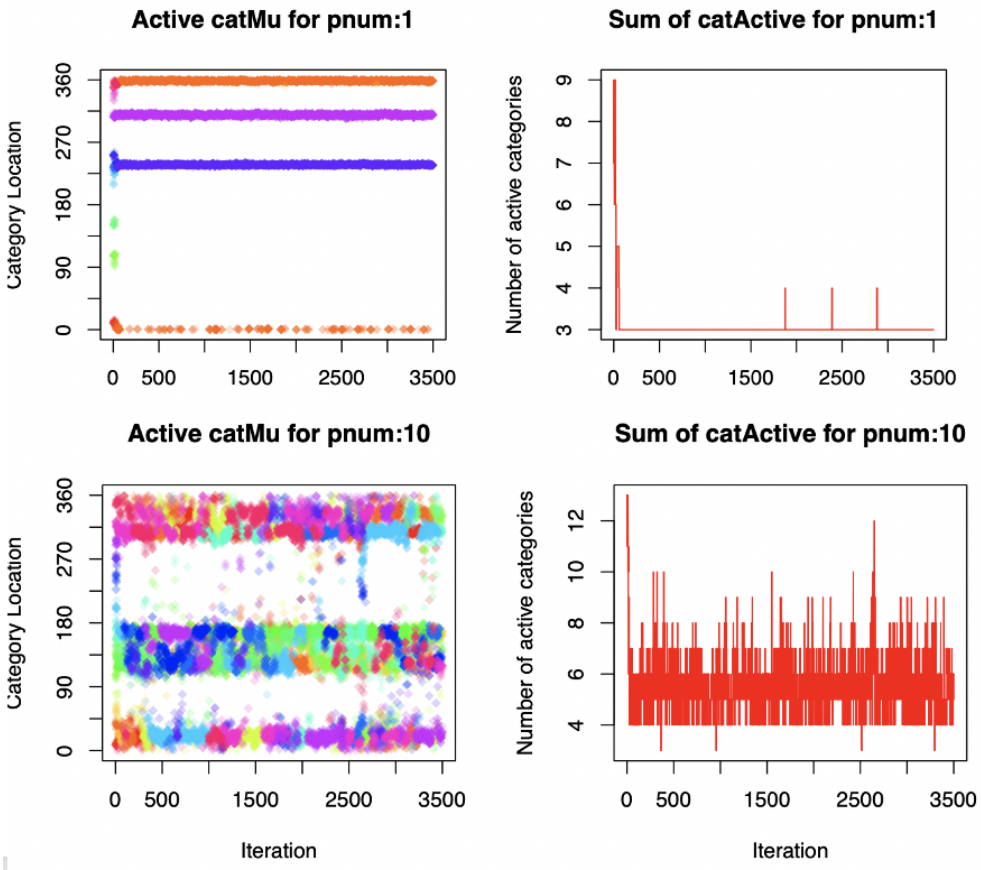


Figure 1: Good Example2

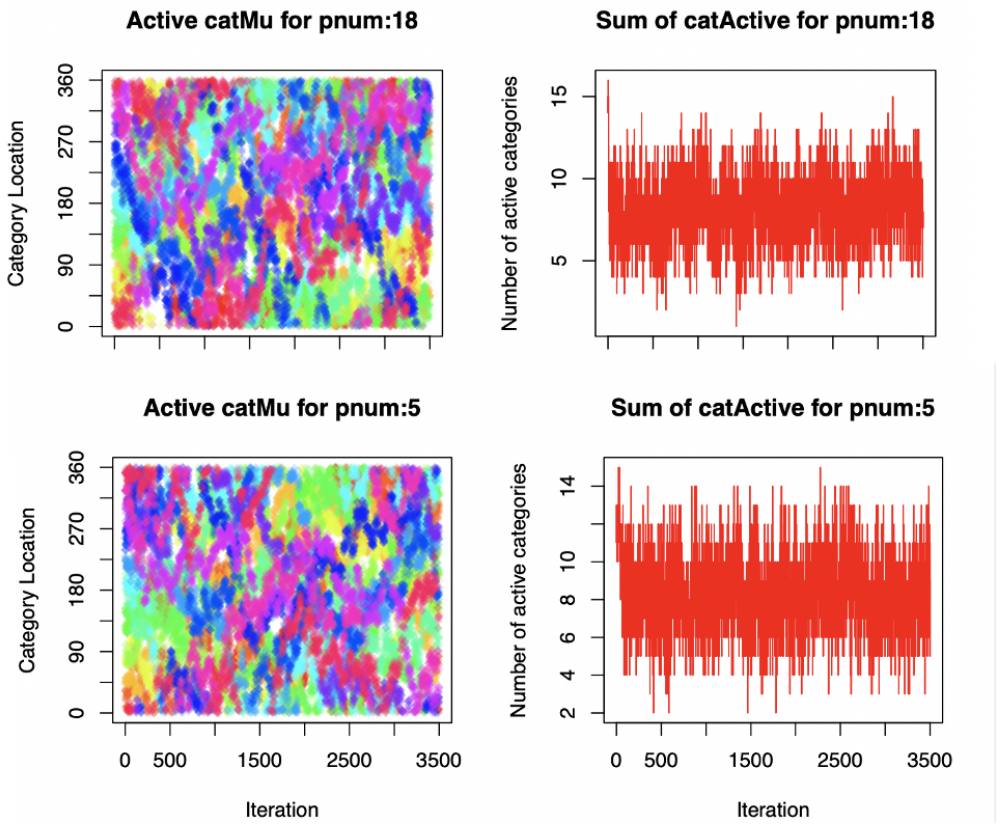


Figure 2: Bad Example2

3.3 Conduct WAIC for model comparison

Model Fit Statistics (WAIC) Models are often compared to one another with model fit statistics like AIC and BIC. For the kinds of Bayesian hierarchical models fit with this package, an appropriate fit statistic is WAIC. WAIC is conceptually similar to AIC, but cannot be directly compared with AIC values (although they do often end up being fairly similar in magnitude).

After making sure that all parameters are convergent in the model, we could try model comparison with calculating WAIC.

How to interpret results: The Model with a smaller WAIC value would be a better model to choose.

Note

To be concluded, we provided guidance on the expand use of CatContModel version 0.8.0. Thanks to Dr. Hardman for his help.