Statistics 243 Final project

Renata Barreto-Montenegro, Aniket Kesari, Aziz Khiyami, AbdulRahman Kreidieh

December 2nd, 2016

1 Main Function

This function takes a dataset, an array with columns grouped into "sets" that correspond to each assessor, and optional arguments for the number of components to be extracted, centering, and scaling. It returns an object of class "mfa" with the specified elements.

```r
mfa <- function(data, sets, ncomps = NULL, center = TRUE, scale = TRUE) {
  # Stop condition
  if (class(data)!="matrix" && class(data)!="data.frame") {
    stop('data must be of class "matrix" or "data.frame"')
  }

  # remove NA's
  data = na.omit(data)

  # Populate "indices" with the groupings from the "sets" argument, and load into a dataframe
  indices = c()
  for (i in 1:length(sets)) {
    indices = c(indices, sets[[i]])
  }
  dat = data[,indices]

  # center and scale if requested
  if (scale) {
    if (center) {
      dat = scale(dat, center = TRUE, scale = FALSE)
      dat = apply(dat, 2 , function(x){x/sqrt(sum(x^2))})
    }
    else {
      dat = scale(dat, center = FALSE, scale = apply(dat, 2, sd, na.rm = TRUE))
    }
  }
  if (center) {
    dat = scale(dat, center = TRUE, scale = FALSE)
  }

  # Step 1: PCA of Each Data Table
  F_partial = list() # Empty list for partial factor
  a = c() # Empty list for alphas
  K = length(sets) # Empty list of length sets for Ks
  J = c() # Empty vector for Js
  index = 1 # Start index at 1
  for (i in 1:length(sets)) {

    # Break up data into each assessor
    Xi = dat[,index:(index+length(sets[[i]])-1)]
    J = c(J, length(sets[[i]]))
  # Compute SVD using the "svd" function in the base package
  SVD = svd(Xi) # Save results of svd on the table of assessors
  U = SVD$u # Pulls a matrix whose columns contain the left singular values
  D = diag(SVD$d) # Pulls a vector containing the singular values of x
  V = SVD$v # Pulls a matrix whose columns contain the right singular vectors of x

  # alpha weights
  alpha_1 = D[1,1]^-2 # Weight is equal to the reciprocal square
  a = c(a, rep(alpha_1,length(sets[[i]]))) # Populate a vector containing the weights for all of the 

  # Partial factor scores (step 1)
  F_partial[[i]] = K*alpha_1*Xi # Take the K value, weight it, and multiply by each assessor's scores

  index = index + length(sets[[i]])
} # Increment the index by the length of the sets

# Step 2: Generalized SVD of X
m = rep(1/dim(dat)[1], dim(dat)[1]) # Table of dimensions

# Compute GSVD
GSVD = svd(diag(m^(1/2)) %*% dat %*% diag(a^(1/2))) # Apply the constraints to the matrix decompositi
Q = t(GSVD$v) %*% diag(a^(-1/2))  # factor loadings

# Eigenvalues
eigenvalues = GSVD$d^2 # Pulls the diagonal vector, and squares it (equal to the vector of eigenvalue

# Common Factor Scores
F_common = dat %*% diag(a) %*% t(Q)

# Partial Factor Scores (step 2)
index = 1
for (i in 1:length(sets)) {
  F_partial[[i]] = F_partial[[i]] %*% t(Q)[index:(index+length(sets[[i]])-1),]
  index = index + length(sets[[i]])
} # Populate partial factor matrix

# Extract number of requested components
if (is.null(ncomps)) {
  ncomps = length(eigenvalues)
}

for (i in 1:length(F_partial)) {
  F_partial[[i]] = F_partial[[i]][,1:ncomps]
}

# Place the results into a list and set the class of the list as "mfa"
res <- list(
  alpha_weights = a,
  Jk = J,
  eigenvalues = eigenvalues[1:ncomps],
  common_factor_scores = F_common[,1:ncomps],
  partial_factor_scores = F_partial,
  factor_loadings = t(Q)[,1:ncomps]
  )
  class(res) <- "mfa"

  return(res)
}
```

1.1 Test the MFA function

Next, we test our MFA function with the wines.csv dataset:

```
# Load data
require(RCurl)
```

```
## Loading required package: RCurl

## Loading required package: bitops
```

```
data <- read.csv(text=getURL('https://raw.githubusercontent.com/ucb-stat243/stat243-fall-2016/master/pro

# Optionally, save the csv
# write.csv(data, file="wines.csv")

# Rename the columns
colnames(data) <- c("ID", "cat pee", "passion fruit", "green pepper", "mineral", "smoky", "citrus", "tro

# Create a list array of the groupings (provided by the paper)
arrays <- list(c(2:7), c(8:13), c(14:19), c(20:24), c(25:30), c(31:35), c(36:39), c(40:45), c(46:50), c

# Test the function and save the result
winescores <- mfa(data, arrays)
```

2 Printing and Plotting

Here, we provide the printing and plotting auxillary functions. These will later be called on in the Shiny app.

```
# Alpha weights
print.alpha.weights <- function(mfa,...) {
  print(mfa$alpha_weights)
}

# JK
print.jk <- function(mfa,...) {
  print(mfa$Jk)
}

# Eigenvalues
print.eigenvalues <- function(mfa,...) {
  print(mfa$eigenvalues)
}

# Compromise
print.mfa.compromise <- function(mfa,...) {
  print(mfa$common_factor_scores)
}
```

3 Summaries of Eigenvalues

Below, we provide code for summarizing eigenvalues.

```
summaries_of_eigenvalues <- function(object, ...) UseMethod('summaries_of_eigenvalues')

summaries_of_eigenvalues.mfa <- function(object) {
  # variables of interest to be placed in table
  eigenvalues = object$eigenvalues
  singularvalues = eigenvalues^(1/2)
  cumulative_eigenvalues = cumsum(eigenvalues)
  inertia = eigenvalues/sum(eigenvalues) * 100
  cumulative_interia = cumsum(inertia)
  tbl = as.data.frame(rbind(singularvalues, eigenvalues, cumulative_eigenvalues, inertia, cumulative_in

  # printing the table
  tbl
}
```

4 Contributions

Below we provide code for calculating the contributions of observations, variables, and tables to a given dimension. See the associated presentation for details.

4.1 Contribution of Observation to a Given Dimension

```
contribution_of_observation <- function(object, ...) UseMethod(contribution_of_observation)

contribution_of_observation.mfa <- function(object, observation_num, dim_num) {
  # the mass of each observation is equal to 1/(number of observers)
  m = 1/length(object$partial_factor_scores)
  f = object$common_factor_scores[observation_num, dim_num]
  lambda = object$eigenvalues[dim_num]

  return(m*f/lambda)
}
```

4.2 Contribution of Variable to a Given Dimension

```
contribution_of_variable <- function(object, ...) UseMethod(contribution_of_variable)

contribution_of_variable.mfa <- function(object, variable_num, dim_num) {
  a = object$alfa_weights[variable_num]
  q = object$factor_loadings[variable_num, dim_num]
  return(a*q)
}
```

4.3 Contribution of Table to a Given Dimension

```
contribution_of_table <- function(object, ...) UseMethod(contribution_of_table)

contribution_of_table.mfa <- function(object, table_num, dim_num) {
  res = 0
  for (i in 1:object$Jk[table_num]) {
    res = res + contribution_of_variable(object, i, dim_num)
  }

  return(res)
}
```

5 Coefficients

5.1 RV Coefficient

Below, we provide code for calculating the R_v coefficient

```
Rv_coefficient <- function(dataset, sets) {
  return(RV_table(object, sets))
}

RV <- function(table1, table2) {
  X_k_k = (table1 %*% t(table1)) %*% (table1 %*% t(table1))
  X_k_kp = (table1 %*% t(table1)) %*% (table2 %*% t(table2))
  X_kp_kp = (table2 %*% t(table2)) %*% (table2 %*% t(table2))

  res = sum(diag(X_k_kp))/sqrt(sum(diag(X_k_k)) * sum(diag(X_kp_kp)))
  return(res)
}

RV_table <- function(dataset, sets) {
  res = matrix(rep(0,length(sets)^2), nrow = length(sets), ncol = length(sets))

  for (i in 1:length(sets)) {
    for (j in 1:i) {
      res[i,j] = RV(dataset[,sets[[i]]], dataset[,sets[[j]]])
      res[j,i] = res[i,j]
    }
  }

  return(res)
}
```

5.2 Lg Coefficient

Below, we provide code for calculating the L_g coefficient

```r
Lg_coefficient <- function(dataset, sets) {
  return(Lg_table(dataset, sets))
}

Lg <- function(table1, table2, alfa) {
  X_k_kp = (table1 %*% t(table1)) %*% (table2 %*% t(table2))

  SVD1 = svd(table1)
  alfa_k = SVD1$d[1]^-2

  SVD2 = svd(table2)
  alfa_kp = SVD2$d[1]^-2

  res = sum(diag(X_k_kp)) * alfa_k * alfa_kp
  return(res)
}

Lg_table <- function(dataset, sets) {
  res = matrix(rep(0,length(sets)^2), nrow = length(sets), ncol = length(sets))

  for (i in 1:length(sets)) {
    for (j in 1:i) {
      res[i,j] = Lg(dataset[,sets[[i]]], dataset[,sets[[j]]])
      res[j,i] = res[i,j]
    }
  }

  return(res)
}
```

5.3 Appendix: Alternative Plotting Methods

```r
## Plot Compromise
plot.mfa.compromise <- function(compromise, dimone, dimtwo, ...) {
  plot(compromise[,dimone],compromise[,dimtwo])
}

## Partial Factor
print.mfa.fpart <- function(mfa,...) {
  print(data.frame(mfa$partial_factor_scores))
}

## Plot Partial Factor
plot.mfa.fpart <- function(fpart, dimone, dimtwo,...) {
  plot(fpart[,dimone], fpart[,dimtwo])
}

## Loading Factor
print.mfa.factor.load <- function(mfa,...) {
  print(mfa$factor_loadings)
}

## Plot Loading Factor
plot.mfa.factor.load <- function(factorload,...) {
  plot(factorload[,dimone], factorload[,dimtwo])
}

# plot function
plot.mfa <- function(x, factor_text = NULL, load_text = NULL, table_text = NULL, ...) {
  # bar chart for the eigenvalues
  barplot(x$eigenvalues, main = 'Eigenvalues', xlab = 'Components', ylab = "Eigenvalue of Component")

  # scatterplot for common factor scores
  plot(x$common_factor_scores[,1], x$common_factor_scores[,2], main = "Common Factor Scores",
       xlab = "F_common[,1]", ylab = "F_common[,2]", type = "n")

  if (is.null(factor_text)) {
    factor_text = c()
    for (i in 1:dim(x$common_factor_scores)[1]) {
      factor_text = c(factor_text, paste('Sample',i))
    }
  }

  text(x$common_factor_scores[,1], x$common_factor_scores[,2], labels = factor_text)

# scatterplot for partial factor scores
  plot(x$common_factor_scores[,1], x$common_factor_scores[,2], main = "Partial Factor Scores",
       xlab = "F_partial[,1]", ylab = "F_partial[,2]", type = "n")

  text(x$common_factor_scores[,1], x$common_factor_scores[,2], labels = factor_text)

  for (i in 1:length(x$partial_factor_scores)) {
    points(x$partial_factor_scores[[i]][,1], x$partial_factor_scores[[i]][,2],
           pch = 16, col = "black")
  }

  # scatterplot for loadings
  if (is.null(load_text)) {
    load_text = c()
    for (i in 1:dim(x$factor_loadings)[1]) {
      load_text = c(load_text, paste('Feature',i))
    }
  }

  plot(x$factor_loadings[,1], x$factor_loadings[,2], main = "Factor Loads",
       xlab = "Load[,1]", ylab = "Load[,2]", type = "n")
  text(x$factor_loadings[,1], x$factor_loadings[,2], labels = load_text)
}
```

6 Shiny App

Here we provide the UI and server code for creating a shiny app that visualizes plots of the eigenvalues, common factor scores, partial factor scores, and factor loadings. For the latter three, the user can specify columns to use for the x and y-axes.

UI:

```
library(shiny)

# Define UI for application that draws a histogram
pageWithSidebar( # Specify format of the app

  # Application title
  titlePanel("MFA"),

  # Sidebar panel that takes three inputs (what to plot, x-axis, and y-axis)
  sidebarPanel(
    selectInput("plot", "Select What to Plot", c("eigenvalue", "common factor scores", "partial factor
    selectInput("col1", "x-axis", c(1:15)),
    selectInput("col2", "y-axis", c(1:15))
    ),

  # Main panel that has the plot
  mainPanel(
    plotOutput("MFAPlot")
  )
)
```

Server:

```
shinyServer(function(input, output) {
  output$MFAPlot <- renderPlot({ # Load in mfa function + data into shiny function
    data <- read.csv(text=getURL('https://raw.githubusercontent.com/ucb-stat243/stat243-fall-2016/maste
    mfa <- function(data, sets, ncomps = NULL, center = TRUE, scale = TRUE) {
      # Stop condition
      if (class(data)!="matrix" && class(data)!="data.frame") {
        stop('data must be of class "matrix" or "data.frame"')
      }

      # remove NA's
      data = na.omit(data)

      indices = c()
      for (i in 1:length(sets)) {
        indices = c(indices, sets[[i]])
      }
      dat = data[,indices]

      # center and scale if requested
      if (scale) {
        if (center) {
          dat = scale(dat, center = TRUE, scale = FALSE)
          dat = apply(dat, 2 , function(x){x/sqrt(sum(x^2))})
        }
```

```r
    else {
      dat = scale(dat, center = FALSE, scale = apply(dat, 2, sd, na.rm = TRUE))
    }
  }
  if (center) {
    dat = scale(dat, center = TRUE, scale = FALSE)
  }


  # Step 1: PCA of Each Data Table
  F_partial = list()
  a = c()
  K = length(sets)
  J = c()
  indx = 1
  for (i in 1:length(sets)) {

    # break up data into each assessor
    Xi = dat[,indx:(indx+length(sets[[i]])-1)]
    J = c(J, length(sets[[i]]))

    # compute SVD
    SVD = svd(Xi)
    U = SVD$u
    D = diag(SVD$d)
    V = SVD$v

    # alfa weights
    alfa_1 = D[1,1]^-2
    a = c(a, rep(alfa_1,length(sets[[i]])))

    # partial factor scores (step 1)
    F_partial[[i]] = K*alfa_1*Xi

    indx = indx + length(sets[[i]])
  }


  # Step 2: Generalized SVD of X
  m = rep(1/dim(dat)[1], dim(dat)[1])

  # compute GSVD
  GSVD = svd(diag(m^(1/2)) %*% dat %*% diag(a^(1/2)))
  Q = t(GSVD$v) %*% diag(a^(-1/2))  # factor loadings

  # eigenvalues
  eigenvalues = GSVD$d^2

  # common factor scores
  F_common = dat %*% diag(a) %*% t(Q)

  # partial factor scores (step 2)
  indx = 1
  for (i in 1:length(sets)) {
    F_partial[[i]] = F_partial[[i]] %*% t(Q)[indx:(indx+length(sets[[i]])-1),]
    indx = indx + length(sets[[i]])
  }

  # in order to extract number of requested components
  if (is.null(ncomps)) {
    ncomps = length(eigenvalues)
  }

  for (i in 1:length(F_partial)) {
    F_partial[[i]] = F_partial[[i]][,1:ncomps]
  }

  # placing results into a list and setting the class of the list as "mfa"
  res <- list(
    alfa_weights = a,  # ask, maybe remove
    Jk = J,  # ask, maybe remove
    eigenvalues = eigenvalues[1:ncomps],
    common_factor_scores = F_common[,1:ncomps],
    partial_factor_scores = F_partial,
    factor_loadings = t(Q)[,1:ncomps]
  )
  class(res) <- "mfa"

  return(res)
}

print.alpha.weights <- function(mfa,...) {
  print(mfa$alfa_weights)
}

print.jk <- function(mfa,...) {
  print(mfa$Jk)
}

print.eigenvalues <- function(mfa,...) {
  print(mfa$eigenvalues)
}

print.mfa.compromise <- function(mfa,...) {
  print(mfa$common_factor_scores)
}

plot.mfa.compromise <- function(compromise, dimone, dimtwo, ...) {
  plot(compromise[,dimone],compromise[,dimtwo])
}

print.mfa.fpart <- function(mfa,...) {
  print(data.frame(mfa$partial_factor_scores))
}

plot.mfa.fpart <- function(fpart, dimone, dimtwo,...) {
      plot(fpart[,dimone], fpart[,dimtwo])
    }

    print.mfa.factor.load <- function(mfa,...) {
      print(mfa$factor_loadings)
    }

    plot.mfa.factor.load <- function(factorload,...) {
      plot(factorload[,dimone], factorload[,dimtwo])
    }

    arrays <- list(c(2:7), c(8:13), c(14:19), c(20:24), c(25:30), c(31:35), c(36:39), c(40:45), c(46:50
    winescores <- mfa(data, arrays)

    x <- as.numeric(input$col1) # Store the values from the first plotting input
    y <- as.numeric(input$col2) # Store the values from the second plotting input

# A series of if statements that tell the app what to plot depending on which type of plot the user sel

    if (input$plot == "eigenvalue") {
    hist(winescores$eigenvalues, breaks=100, main="Histogram of Eigenvalues", xlab="Eigenvalue", ylab="
  }
    if (input$plot == "common factor scores") {
      plot(winescores$common_factor_scores[,x], winescores$common_factor_scores[,y], main="Common Facto
    }
    if (input$plot == "partial factor scores") {
      X <- print.mfa.fpart(winescores)
      plot.mfa.fpart(X, x, y)
    }
    if (input$plot == "loadings factor scores") {
      plot(winescores$factor_loadings[,x], winescores$factor_loadings[,y])
    }
  }
  )
})
```