An Introduction to cpfa

Matthew Snodgress

University of Minnesota, May 16, 2022

Overview

Package **cpfa** implements a k-fold cross-validation procedure to predict class labels using component loadings from a single mode of a Parallel Factor (Parafac) model-1 (Harshman, 1970; Harshman & Lundy, 1994), which is fit to a three-way or four-way data tensor. After fitting a Parafac model with package **multiway** via an alternating least squares algorithm (Helwig, 2019), estimated component loadings from one mode of this model are passed to one or several classification methods. These methods and the packages used to implement them include: penalized logistic regression (PLR) implemented through **glmnet** (Zou & Hastie, 2005; Friedman, Hastie, & Tibshirani, 2010); support vector machine (SVM) implemented through **e1071** (Cortes & Vapnik, 1995; Meyer et al., 2021); random forest (RF) through **randomForest** (Breiman, 2001; Liaw & Wiener, 2002); and feed-forward neural network (NN) through **nnet** (Ripley, 1994; Venables & Ripley, 2002). For each method, a k-fold cross-validation is conducted to tune classification parameters using estimated Parafac component loadings, optimizing class label prediction. Multiple constraint options are available to impose on any mode of the Parafac model during the estimation step (see Helwig, 2017a). Multiple numbers of components can be considered over multiple Parafac models in the primary package function `cpfa`. This vignette describes the specific procedure implemented in **cpfa** and its usage in R (R Core Team, 2022).

Introduction

A bilinear model, such as principal components analysis (PCA) or independent components analysis (ICA), assumes that a data matrix can be decomposed into R underlying components (or factors), which consist of an outer product of a row mode (A mode) and a column mode (B mode), respective to the matrix structure. Let $\mathbf{X}$ be the $I \times J$ data, for $i = 1, 2, \ldots, I$, and for $j = 1, 2, \ldots, J$. A bilinear model decomposes $\mathbf{X}$ such that

$$\mathbf{X} = \mathbf{A}\mathbf{B}^\top + \mathbf{E} \tag{1}$$

where $\mathbf{A}$ is the $I \times R$ matrix of mode A weights on R underlying components, and $\mathbf{B}$ is the $J \times R$ matrix of mode B weights on R components. The $I \times J$ matrix $\mathbf{E}$ contains error not explained by the model. Then, given K levels of a third mode, for $k = 1, 2, \ldots, K$, for some data tensor $\underline{\mathbf{X}}$ of dimensions $I \times J \times K$, the Parafac model can be written as

$$\mathbf{X}_k = \mathbf{A}\mathbf{C}_k\mathbf{B}^\top + \mathbf{E}_k \tag{2}$$

where $\mathbf{X}_k$ is the k-th level's $I \times J$ data matrix. The $\mathbf{A}$ and $\mathbf{B}$ matrices have the same interpretation as in the bilinear model, and $\mathbf{C}_k$ is an $R \times R$ diagonal matrix with the k-th level's component scores along its diagonal. The $\mathbf{E}_k$ matrix is the error matrix for the k-th level, which is the portion of $\mathbf{X}_k$ that cannot be explained by the Parafac model structure (Harshman, 1970). The Parafac model parameters can be estimated using an alternating least squares (ALS) algorithm with multiple random starts (Kiers, Ten Berge, & Bro, 1999; Faber, Bro, & Hopke, 2003; Tomasi & Bro 2006). Constraints can be included on any mode's weight matrix (Helwig, 2017a). For added constraints, the corresponding step of the ALS algorithm is replaced with a constrained least squares update. Note that this Parafac model is directly generalizable to data tensors of higher dimensions (e.g., a four-way data tensor of dimensions $I \times J \times K \times L$).

Consider the following classification problem: either a three-way data tensor of dimension $I \times J \times K$ or a four-way data tensor of dimension $I \times J \times K \times L$, denoted by $\underline{\mathbf{X}}$, is used to predict a group label vector $\mathbf{y}$ of dimension $N \times 1$ where N is equal to at least one of the data tensor dimensions (e.g., $N = I$ or $N = K$). Both $\underline{\mathbf{X}}$ and $\mathbf{y}$ are randomly split into training and testing sets using some split ratio (e.g., 80/20 split), producing: (1) a training data tensor $\underline{\mathbf{X}}_{\text{train}}$; (2) a testing data tensor $\underline{\mathbf{X}}_{\text{test}}$; (3) a training label vector $\mathbf{y}_{\text{train}}$; and (4) a testing label vector $\mathbf{y}_{\text{test}}$. Note that $\underline{\mathbf{X}}$ is split along a single mode with the number of levels equal to N, which we call the classification mode.

Using these split data, **cpfa** implements an analysis procedure in four steps. Consider the case for a three-way data tensor where $N = K$ such that the third mode is the classification mode.

Step 1 - Fit Parafac Model: The Parafac model is fit to the training data tensor $\underline{\mathbf{X}}_{\text{train}}$ to obtain estimates of the component weights for the first mode $\hat{\mathbf{A}}_{\text{train}}$, second mode $\hat{\mathbf{B}}_{\text{train}}$, and third mode $\hat{\mathbf{C}}_{\text{train}}$. A Parafac model can be fit multiple times over different combinations of R (e.g, for $R \in \{1, 2, 3, \ldots, 5\}$) and over different combinations of mode constraints (e.g., unconstrained on all three modes, orthogonal $\mathbf{A}$, or smoothness on $\mathbf{A}$ and on $\mathbf{B}$). Run `const()` to see all possible options.

Step 2 - Predict Features: For any fit Parafac model, the estimated Parafac $\hat{\mathbf{A}}_{\text{train}}$ and $\hat{\mathbf{B}}_{\text{train}}$ are used to estimate test set features $\hat{\mathbf{C}}_{\text{test}}$ from the test set tensor $\underline{\mathbf{X}}_{\text{test}}$. The least squares estimates of the features have the form $\hat{\mathbf{C}}_{\text{test}} = \mathbf{X}^*_{\text{test}}\mathbf{Z}_{\text{train}}(\mathbf{Z}^\top_{\text{train}}\mathbf{Z}_{\text{train}})^{-1}$, where $\mathbf{X}^*_{\text{test}}$ is the tensor $\underline{\mathbf{X}}_{\text{test}}$ arranged into a matrix of dimension $K \times (I \times J)$, and $\mathbf{Z}_{\text{train}} = \hat{\mathbf{A}}_{\text{train}} \odot \hat{\mathbf{B}}_{\text{train}}$ denotes the Khatri-Rao product (i.e., columnwise Kronecker product) between the Parafac first mode and second mode estimated from the training data.

Step 3 - Train Classifier: For any fit Parafac model, the estimated third mode weights $\hat{\mathbf{C}}_{\text{train}}$ are used as features to classify the group labels $\mathbf{y}_{\text{train}}$. Four classification methods are possible: PLR, SVM, RF, and NN. For each classification method, observations can be weighted to account for class imbalance, and either one or two classification parameters are tuned via k-fold cross-validation to identify the values that minimize the expected misclassification rate. Users can supply ranges for the classification parameters for any of the four methods. For PLR, **cpfa** tunes α with the penalty parameter λ chosen by `cv.glmnet()`. For SVM, **cpfa** assumes a radial basis kernel function and tunes two parameters: cost and γ. For RF, the number of splitting predictors is fixed at $\sqrt{R}$ and both the number of trees and the minimum node size can be tuned. Finally, for NN, classification parameters size (number of hidden layer units) and decay (weight decay) can be tuned (see Appendix for model details for all four methods). The function `cpfa` implements Steps 1 - 3.

Step 4 - Evaluate Classifier: The predicted features from Step 2 are input as features to the trained classifier from Step 3 to obtain the test set classifications $\hat{\mathbf{y}}_{\text{test}}$ using `predict.cpfa`. These classifications are used to calculate any of 11 possible classification performance measures calculated with function `cpm`, which can compare $\hat{\mathbf{y}}_{\text{test}}$ to $\mathbf{y}_{\text{test}}$.

Note that the function `cpfa` contains an argument to implement parallel computing through packages **parallel** and **doParallel** (R Core Team, 2022; Microsoft Corporation & Steve Weston, 2022). The package **cpfa** also contains a function `print.cpfa` for printing results from objects produced by `cpfa`.

Installation

cpfa can be installed directly from CRAN. Type the following command in a R console:

```r
install.packages("cpfa", repos = "https://cran.us.r-project.org")
```

The argument `repos` can be modified according to user preferences. For more options and details, see `help(install.packages)`. In this case, the package **cpfa** has been downloaded and installed to the default directories. Users can download the package source at https://cran.r-project.org/package=cpfa and use Unix commands for installation.

Example 1: Three-way Tensor with Binary Response

We start by explaining the main function `cpfa` and examining basic operations and outputs related to this function.

First, we load the **cpfa** package:

```r
library(cpfa)
```

```
## Loading required package: multiway

## Loading required package: CMLS

## Loading required package: quadprog

## Loading required package: parallel

## Loading required package: glmnet

## Loading required package: Matrix

## Loaded glmnet 4.1-4

## Loading required package: e1071

## Loading required package: randomForest

## randomForest 4.7-1

## Type rfNews() to see new features/changes/bug fixes.

## Loading required package: doParallel

## Loading required package: foreach

## Loading required package: iterators
```

We load a data set found within **cpfa** for this example. Users can either use example data created below or load their own data. We start by creating a random three-way tensor and response vector where one mode (a classification mode) is related to the response, in this case the third mode. To generate the data, we identify a target correlation matrix that sets the correlations among the columns of the classification mode's weight matrix and between these weight matrix columns and a response vector. We then use a Cholesky decomposition to generate random data that would give rise to this correlation matrix and modify that data to generate a tensor related to the response (see Gentle, 1998; see also examples under `help(parafac)`). In R:

```r
# create random data for three-way tensor with Parafac structure and response
set.seed(123)
mydim <- c(32, 25, 240)
nfac <- nf <- 3
rho.c.c <- .35
rho.c.y <- .75
R <- matrix(c(  1, rho.c.c, rho.c.c, rho.c.y,
                 rho.c.c, 1, rho.c.c, rho.c.y,
                 rho.c.c, rho.c.c, 1, rho.c.y,
                 rho.c.y, rho.c.y, rho.c.y, 1), nrow = nfac+1, ncol = nfac+1)
Nsubj <- mydim[3]
Nvar <- nfac + 1
C.col <- runif(Nsubj*nfac)
y.col <- rbinom(Nsubj, 1, 0.5)
values <- c(C.col, y.col)
Y <- matrix(values, nrow = Nsubj, ncol = Nvar)
Y <- Y - matrix(1, Nsubj, 1) %*% apply(Y, 2, mean)
S <- t(Y) %*% Y
M <- t(chol(S))
Minv <- solve(M)
L <- t(chol(R))
Xq <- Y %*% t(Minv) %*% t(L)
Cmat <- Xq[, 1:3]*2
Amat <- matrix(rnorm(mydim[1]*nfac), nrow = mydim[1], ncol = nfac)
Bmat <- matrix(runif(mydim[2]*nfac), nrow = mydim[2], ncol = nfac)
Xmat <- tcrossprod(Amat, krprod(Cmat, Bmat))
Xmat <- array(Xmat, dim = mydim)
Emat <- array(rnorm(prod(mydim)), dim = mydim)
Emat <- nscale(Emat, 0, ssnew = sumsq(Xmat))
X <- Xmat + Emat
y <- factor(as.numeric(Xq[, 4] > 0))
```

The above creates a random input tensor `X` and a response vector `y`. We confirm the dimensions of `X` and `y`, their classes, and the possible values of `y`:

```
# examine data object X
dim(X)
```

```
## [1]  32  25 240
```

```
class(X)
```

```
## [1] "array"
```

```
# examine data object y
class(y)
```

```
## [1] "factor"
```

```
length(y)
```

```
## [1] 240
```

```
table(y)
```

```
## y
##   0   1
## 117 123
```

Input `X` has dimensions $32 \times 25 \times 240$ for the first, second, and third modes, respectively, and is of class "array". The main function `cpfa` requires an input data tensor to be of class "array" with either three modes or four modes, which is satisfied. Response vector `y` is of class "factor", which `cpfa` requires as the class for a response input. Moreover, the length of `y` must match the length of one of the dimensions of `X`; and response `y` has 240 values. The third mode of `X` equals the length of response `y` (i.e., both are equal to 240); and we assume the third mode of `X` is then the classification mode. By default, function `cpfa` assumes the classification mode is the third mode (or fourth for a four-way tensor input). Argument `cmode` in `cpfa` can be used to specify a different classification mode if the input array is not organized already with the classification mode last (see Example 2 for a use of `cmode`).

The function `table` also indicates that `y` contains two class labels, showing that `y` contains a binary response. There appears to be a slight imbalance in the observed class memberships. Function `cpfa` calculates the observed proportions of class memberships in `y` and provides this information to classification methods to account for imbalanced data when training classifers (see Introduction, Step 3 for classifier training information). If this default is undesirable, users can specify prior probabilities of class membership using the argument `prior` in function `cpfa`. See `help(cpfa)`. We ignore this argument for now.

We next split our data into training and testing sets using an 80/20 split (80% in training, and 20% in testing):

```
# split data into training and testing sets
split <- 0.8
nlev <- length(y)
index <- round(split*nlev)
X.train <- X[, , 1:index]
X.test <- X[, , (index+1):nlev]
y.train <- y[1:index]
y.test <- as.numeric(y[(index + 1):nlev]) - 1
```

We specify (1) the number of components to be used when estimating multiple Parafac models using argument `nfac` (see Introduction, Step 1); (2) the number of folds for k-fold cross-validation via the argument `nfolds` (see Introduction, Step 3) ; and (3) a set of fold IDs for each observation specified with argument `foldid`. Argument `method` identifies the classifiers to be trained. We set `method` to include all four of the possible classification methods (see below). Users can set ranges for tuned classification parameters; we provide a range for RF parameter `ntree` as an example (see `help(cpfa)` for a full list). We also indicate that the classification response is binary with `family <- "binomial"` and that the computation is not to be parallelized via `parallel <- FALSE`. Note that setting `parallel` to `TRUE` will initiate parallel computing. `cpfa` will also generate a cluster `cl` via `cl <- makeCluster(detectCores())` if no cluster is provided and `parallel <- TRUE`. A cluster can be provided via the `cl` argument passed directly to **multiway** (see `help(parafac)` for more information).

We set several arguments to be passed directly to package **multiway** when fitting Parafac models (Introduction, Step 1). Specifically, we set the constraints to be used for each of the three modes of the Parafac model, which include: an orthogonality constraint for the first mode, a smoothness constraint for the second mode, and no constraints for the third mode. For the ALS algorithm, we set the tolerance threshold via `ctol <- 1e-02` and the number of random starts via `nstart <- 5` (see `help(parafac)`). Note that we would likely set `ctol` to a lower value for real applications and here set it to a larger value in order to improve computation speed for this example. Likewise, we set `nstart` to a lower value than might be needed for real applications. In R:

```
# initialize inputs for cpfa
nfac <- 1:3
nfolds <- 5
foldid <- sample(rep(1:nfolds, length.out = length(y.train)))
method <- c("PLR", "SVM", "RF", "NN")
ntree <- c(100, 300, 500)
family <- "binomial"
parallel <- FALSE

# initialize inputs passed directly to 'parafac' within package 'multiway'
const <- c("orthog", "smooth", "uncons")
ctol <- 1e-02
nstart <- 5
```

We input the above into main function `cpfa` and run the function:

```
tune.object <- cpfa(x = X.train, y = y.train, nfac = nfac, nfolds = nfolds,
                    foldid = foldid, method = method, ntree = ntree,
                    family = family, parallel = parallel, const = const,
                    ctol = ctol, nstart = nstart)
```

```
## nfac = 1 method = parafac
##   |                                                                          |
## nfac = 1 method = plr
## nfac = 1 method = svm
## nfac = 1 method = rf
## nfac = 1 method = nn
## nfac = 2 method = parafac
##   |                                                                          |
## nfac = 2 method = plr
## nfac = 2 method = svm
## nfac = 2 method = rf
## nfac = 2 method = nn
## nfac = 3 method = parafac
##   |                                                                          |
## nfac = 3 method = plr
## nfac = 3 method = svm
## nfac = 3 method = rf
## nfac = 3 method = nn
```

Note that progress is listed for (a) the number of factors/components in the current Parafac model being fit and for (b) the method currently in progress. Progress listing can be turned off by setting argument `verbose` to `FALSE`.

The output is stored in object `tune.object`, which is of class `cpfa`. Running the object's name prints a summary of the tuning process conducted in function `cpfa`. In R:

```
tune.object
```

```
## Parafac Models Estimated:
## 3-way Parafac with 1 factors
## 3-way Parafac with 2 factors
## 3-way Parafac with 3 factors
##
## Classification Methods Tuned:
## PLR
## SVM
## RF
## NN
##
## KCV Misclassification Error (estimation time in seconds) by
## Model and Method:
##
## Parafac with 1 factors:
##   PLR:  Error = 0.0514 (0.319)
##   SVM:  Error = 0.0517 (1.371)
##   RF:   Error = 0.0622 (1.977)
##   NN:   Error = 0.0622 (2.849)
## Parafac with 2 factors:
##   PLR:  Error = 0.0306 (0.284)
##   SVM:  Error = 0.0309 (1.399)
##   RF:   Error = 0.036 (2.183)
##   NN:   Error = 0.0309 (3.395)
## Parafac with 3 factors:
##   PLR:  Error = 0.0306 (0.29)
##   SVM:  Error = 0.031 (1.491)
##   RF:   Error = 0.0464 (2.417)
##   NN:   Error = 0.0258 (3.848)
```

The Parafac models that were fit and the classifiers trained are both listed. Also listed is the k-cv misclassification error (i.e., averaged over the number of folds) for each model and method. The approximate estimation time is listed in parentheses in seconds next to each classifier under each model.

The output stores a number of objects. For example, `tune.object$opt.model` contains a list of the optimal models for each classifier (i.e., with the classification parameters that minimized misclassification error). `tune.object$opt.param` contains a matrix that lists these parameters for each fit Parfac model. Estimated component weight matrices are also provided in `tune.object$Aweights`, `tune.object$Bweights`, and `tune.object$Cweights`. Note that `tune.object$Cweights` is NULL for a three-way tensor and includes estimated third mode weights when a four-way tensor is instead provided (e.g., see Example 2). For a list of all provided output, see `help(cpfa)`.

To determine classification performance, we predict class labels using the testing set (see Introduction, Step 4). We call function `predict.cpfa` by inputting `tune.object` of class `cpfa` into the `predict` function. In R:

```
yhat <- predict(tune.object, newdata = X.test, type = "response")
```

Predicted class membership for each test set observation is found in `yhat` across each Parafac model (i.e., number of factors/components) and each classifier. Alternatively, predicted class membership probabilities can be estimated instead of class membership labels by specifying `type = prob`. If neither labels or probabilities are desired, predicted component loadings can be calculated instead with `type = classify.weights`. We focus on a three-factor Parafac model for classifier RF and use function `cpm` to calculate performance measures:

```
perform.output <- cpm(yhat$fac.3rf, y.test)
perform.output$cm
```

```
##    0  1
## 0 25  1
## 1  1 21
```

```
perform.output$class.eval
```

```
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
```

Function `cpm` creates output with two parts. The first is the confusion matrix `cm` for the classification. The second is a set of classification performance values contained within `class.eval` (see `help(cpm)` for details on these measures). The first performance measure, for example, is the overall classification error. In this example, it appears that the classifier is performing well—assuming for these example data that good accuracy is, say, error (`err`) below 0.25. To examine performance for all numbers of components and all classifiers, we can write:

```
apply(yhat, 2, function(x){cpm(x, y.test)})
```

```
## $fac.1plr
## $fac.1plr$cm
##    0  1
## 0 25  1
## 1  1 21
##
## $fac.1plr$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
##
##
## $fac.1svm
## $fac.1svm$cm
##    0  1
## 0 25  1
## 1  1 21
##
## $fac.1svm$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
##
##
## $fac.1rf
## $fac.1rf$cm
##    0  1
## 0 25  1
## 1  1 21
##
## $fac.1rf$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
##
##
## $fac.1nn
## $fac.1nn$cm
##    0  1
## 0 25  1
## 1  1 21
##
## $fac.1nn$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
##
##
## $fac.2plr
## $fac.2plr$cm
##    0  1
## 0 26  2
## 1  0 20
##
```

```
## $fac.2plr$class.eval
##          err       acc tpr        fpr       tnr fnr       ppv npv        fdr
## 1 0.04166667 0.9583333   1 0.07142857 0.9285714   0 0.9090909   1 0.09090909
##   fom       fs
## 1   0 0.952381
##
##
## $fac.2svm
## $fac.2svm$cm
##    0  1
## 0 26  3
## 1  0 19
##
## $fac.2svm$class.eval
##      err    acc tpr       fpr       tnr fnr       ppv npv       fdr fom
## 1 0.0625 0.9375   1 0.1034483 0.8965517   0 0.8636364   1 0.1363636   0
##          fs
## 1 0.9268293
##
##
## $fac.2rf
## $fac.2rf$cm
##    0  1
## 0 25  2
## 1  1 20
##
## $fac.2rf$class.eval
##      err    acc      tpr        fpr       tnr        fnr       ppv       npv
## 1 0.0625 0.9375 0.952381 0.07407407 0.9259259 0.04761905 0.9090909 0.9615385
##          fdr        fom        fs
## 1 0.09090909 0.03846154 0.9302326
##
##
## $fac.2nn
## $fac.2nn$cm
##    0  1
## 0 26  2
## 1  0 20
##
## $fac.2nn$class.eval
##          err       acc tpr        fpr       tnr fnr       ppv npv        fdr
## 1 0.04166667 0.9583333   1 0.07142857 0.9285714   0 0.9090909   1 0.09090909
##   fom       fs
## 1   0 0.952381
##
##
## $fac.3plr
## $fac.3plr$cm
##    0  1
## 0 26  2
## 1  0 20
##
## $fac.3plr$class.eval
##          err       acc tpr        fpr       tnr fnr       ppv npv        fdr
## 1 0.04166667 0.9583333   1 0.07142857 0.9285714   0 0.9090909   1 0.09090909
##   fom       fs
## 1   0 0.952381
##
##
## $fac.3svm
## $fac.3svm$cm
##    0  1
## 0 26  2
## 1  0 20
##
## $fac.3svm$class.eval
##          err       acc tpr        fpr       tnr fnr       ppv npv        fdr
## 1 0.04166667 0.9583333   1 0.07142857 0.9285714   0 0.9090909   1 0.09090909
##   fom       fs
## 1   0 0.952381
##
##
## $fac.3rf
## $fac.3rf$cm
##    0  1
## 0 25  1
## 1  1 21
##
## $fac.3rf$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9545455 0.03846154 0.9615385 0.04545455 0.9545455
##         npv        fdr        fom        fs
## 1 0.9615385 0.04545455 0.03846154 0.9545455
##
##
## $fac.3nn
## $fac.3nn$cm
##    0  1
## 0 26  2
## 1  0 20
##
## $fac.3nn$class.eval
##          err       acc tpr        fpr       tnr fnr       ppv npv        fdr
## 1 0.04166667 0.9583333   1 0.07142857 0.9285714   0 0.9090909   1 0.09090909
##   fom       fs
## 1   0 0.952381
```

Example 2: Four-way Tensor with Multiclass Response

We consider a second example that includes a four-way input tensor and a response vector with three possible class memberships (a multiclass response). We start by creating a random four-way tensor and response vector where one mode (a classification mode) is related to the response, in this case the third mode:

```
# create random data for four-way tensor with Parafac structure and response
set.seed(123)
nfac <- nf <- 3
mydim <- c(32, 25, 240, 20)
aseq <- seq(-3, 3, length.out = mydim[1])
Amat <- cbind(dnorm(aseq), dchisq(aseq+3.1, df=3),
```

```r
               dt(aseq-2, df=4), dgamma(aseq+3.1, shape=3, rate=1))[,1:3]
Bmat <- svd(matrix(runif(mydim[2]*nf), nrow = mydim[2], ncol = nf), nv = 0)$u
rho.c.c <- .725
rho.c.y <- .9
R <- matrix(c(  1, rho.c.c, rho.c.c, rho.c.y,
                rho.c.c, 1, rho.c.c, rho.c.y,
                rho.c.c, rho.c.c, 1, rho.c.y,
                rho.c.y, rho.c.y, rho.c.y, 1), nrow = nfac+1, ncol = nfac+1)
Nsubj <- mydim[3]
Nvar <- nfac + 1
C.col <- runif(Nsubj*nfac)
y.col <- rbinom(Nsubj, 1, 0.5)
values <- c(C.col, y.col)
Y <- matrix(values, nrow = Nsubj, ncol = Nvar)
Y <- Y - matrix(1, Nsubj, 1) %*% apply(Y, 2, mean)
S <- t(Y) %*% Y
M <- t(chol(S))
Minv <- solve(M)
L <- t(chol(R))
Xq <- Y %*% t(Minv) %*% t(L)
Cmat <- Xq[, 1:3]*2
y <- Xq[,4]
for(i in 1:length(y)){
  if(abs(y[i]) > 0.09){
    y[i] <- 2
  }
  if((y[i] < 0.09) & (y[i] > 0)){
    y[i] <- 1
  }
  if((y[i] > -0.09) & (y[i] < 0)){
    y[i] <- 0
  }
}
Dmat <- matrix(runif(mydim[4]*nf), nrow = mydim[4], ncol = nf)
Xmat <- tcrossprod(Amat, krprod(Dmat, krprod(Cmat, Bmat)))
Xmat <- array(Xmat, dim = mydim)
Emat <- array(rnorm(prod(mydim)), dim = mydim)
Emat <- nscale(Emat, 0, ssnew = sumsq(Xmat))   # SNR = 1
X <- Xmat + Emat
X2 <- X
y2 <- factor(y)
```

We note the dimensions of the four-way data array X2 and its object class:

```r
dim(X2)
```

```
## [1]  32  25 240  20
```

```r
class(X2)
```

```
## [1] "array"
```

In this example, tensor $\underline{\mathbf{X}}$ in object X2 is of dimensions $32 \times 25 \times 240 \times 20$ and is of class “array”, which is required by the main function `cpfa`. We assume the classification mode is the third mode containing 240 levels. We next examine the response vector `y2`:

```
class(y2)
```

```
## [1] "factor"
```

```
length(y2)
```

```
## [1] 240
```

```
table(y2)
```

```
## y2
##  0  1  2 
## 98 98 44
```

The response vector is of class “factor”, which is required by `cpfa`. It contains 240 class labels with three possible values, and there exists some imbalance among the three classes. As before in Example 1, we split the data into training and testing sets using an 80/20 split ratio. Note that we split the input tensor using its third mode as we assume that the third mode is the classification mode:

```
# split data into training and testing sets
split <- 0.8
nlev <- length(y2)
index <- round(split*nlev)
X.train <- X2[, , 1:index, ]
X.test <- X2[, , (index+1):nlev, ]
y.train <- y2[1:index]
y.test <- as.numeric(y2[(index + 1):nlev]) - 1
```

Also as in Example 1, we initialize arguments to input into the main function `cpfa`:

```
# initialize inputs for cpfa
nfac <- 3
nfolds <- 3
foldid <- sample(rep(1:nfolds, length.out = length(y.train)))
method <- c("PLR", "SVM", "RF", "NN")
ntree <- c(100, 300, 500)
family <- "multinomial"
parallel <- FALSE

# initialize inputs passed directly to 'parafac' within package 'multiway'
const <- c("orthog", "smooth", "uncons", "uncons")
ctol <- 1e-02
nstart <- 5
```

Compared to Example 1, we note several changes. First, we only examine a Parafac model with three components/factors in this example. Second, the number of folds in k-fold cross-validation is now set to three while another value is included in the input for argument `const` to designate constraints (or their absence) for each of the four tensor modes. Finally, because the response is multiclass, we have set `family <- multinomial`, which function `cpfa` interprets as a multiclass classification problem. Before inputting into `cpfa`, we also now set argument `cmode`, which designates the classification mode of the input tensor:

```
cmode <- 3
```

In this case, the third mode is the classification mode. Inputting into `cpfa`:

```
tune.object2 <- cpfa(x = X.train, y = y.train, nfac = nfac, nfolds = nfolds,
                     foldid = foldid, method = method, ntree = ntree,
                     family = family, parallel = parallel, const = const,
                     ctol = ctol, nstart = nstart, cmode = cmode)
```

```
## nfac = 3 method = parafac
##   |                                                                              |
## nfac = 3 method = plr
## nfac = 3 method = svm
## nfac = 3 method = rf
## nfac = 3 method = nn
```

Output is similar to output from Example 1 with `tune.object2$Cweights` now containing estimated weights for the third mode. Note that `cpfa` shifts the classification mode to the last (furthest right) mode when argument `cmode` is provided. In this case, input modes ordered 1, 2, 3, and 4 correspond to output modes ordered 1, 2, 4, and 3 (i.e., classification mode has been reordered to be last). We predict class labels using the testing set and examine the classification performance:

```
yhat <- predict(tune.object2, newdata = X.test, type = "response")
apply(yhat, 2, function(x){cpm(x, y.test)})
```

```
## $fac.3plr
## $fac.3plr$cm
##    0  1  2
## 0 15  1 10
## 1  1 18  3
## 2  0  0  0
## 
## $fac.3plr$class.eval
##      err    acc tpr       fpr       tnr fnr       ppv       npv       fdr
## 1 0.3125 0.6875 NaN 0.1325801 0.8674199 NaN 0.6282895 0.8033878 0.3717105
##         fom  fs
## 1 0.1966122 NaN
## 
## 
## $fac.3svm
## $fac.3svm$cm
##    0  1  2
## 0 15  0  0
## 1  0 19  1
## 2  1  0 12
## 
## $fac.3svm$class.eval
##          err       acc       tpr        fpr       tnr        fnr       ppv
## 1 0.04166667 0.9583333 0.9576923 0.01994048 0.9800595 0.04230769 0.9535256
##         npv        fdr        fom        fs
## 1 0.9785714 0.04647436 0.02142857 0.9556044
## 
## 
## $fac.3rf
## $fac.3rf$cm
##    0  1  2
## 0 14  0  1
## 1  0 19  2
## 2  2  0 10
## 
## $fac.3rf$class.eval
##         err       acc       tpr        fpr       tnr       fnr       ppv
## 1 0.1041667 0.8958333 0.8904762 0.04928315 0.9507168 0.1095238 0.8814103
##         npv       fdr        fom      fs
## 1 0.9442002 0.1185897 0.05579976 0.88592
##
##
## $fac.3nn
## $fac.3nn$cm
##    0  1  2
## 0 15  2 10
## 1  1 17  3
## 2  0  0  0
##
## $fac.3nn$class.eval
##         err       acc tpr       fpr       tnr fnr       ppv       npv       fdr
## 1 0.3333333 0.6666667 NaN 0.1540305 0.8459695 NaN 0.6107456 0.7918935 0.3892544
##         fom  fs
## 1 0.2081065 NaN
```

Note that if a confusion matrix contain all zeros on one of its rows or columns (i.e., is singular), some performance measures will return `NaN`.

Concluding Thoughts

Package **cpfa** implements a k-fold cross-validation procedure discussed in the Introduction, connecting Parafac models fit by **multiway** to classification methods implemented through four popular packages used for classification, including: **glmnet**, **e1071**, **randomForest**, and **nnet**. Parallel computing is implemented through packages **parallel** and **doParallel**. The two examples above highlight the use of **cpfa** and its three main functions. For more information about the package, see https://CRAN.R-project.org/package=cpfa or examine package help files with `help(cpfa)`, `help(predict.cpfa)`, or `help(cpm)`. Note that Parafac model details are provided in the Introduction while classification methods are specified further in the Appendix.

Appendix: Classification Methods

Penalized Logistic Regression (PLR)

Binomial logistic regression extends ordinary linear regression to cases with binary responses by placing linear terms within a logistic link function. Given a binary response $y \in \{0,1\}$, the probability of a given class is modeled as

$$P(y=1|\mathbf{x}) = \frac{1}{1+\exp\left(-(\beta_0+\mathbf{x}^\top\boldsymbol{\beta})\right)}, \tag{3}$$

where $\mathbf{x} = (x_1,\ldots,x_p)^\top$ is the observed predictor vector, β_0 is the unknown intercept parameter, and $\boldsymbol{\beta} = (\beta_1,\ldots,\beta_p)^\top$ is the unknown vector of slope parameters. The probability for the remaining class is $P(y=0|\mathbf{x}) = 1 - P(y=1|\mathbf{x})$. Given n independent observations $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, binomial maximum likelihood is used to estimate the parameters. Specifically, the maximum likelihood estimation approach seeks to find the $(\beta_0, \boldsymbol{\beta})$ that maximize the log-likelihood function

$$\ell(\beta_0,\boldsymbol{\beta}) = \sum_{i=1}^{n} -\log(1+\exp(\beta_0+\mathbf{x}_i^T\boldsymbol{\beta})) + \sum_{i=1}^{n} y_i(\beta_0+\mathbf{x}_i^T\boldsymbol{\beta}), \tag{4}$$

for $i = 1,\ldots,n$ (Cox, 1958). In penalized logistic regression (PLR), a penalty is added to the logistic loss function to introduce bias and reduce the estimator's variance. The new objective can be stated

$$\min_{\beta_0,\boldsymbol{\beta}} \left\{ -\ell(\beta_0,\boldsymbol{\beta}) + \lambda \sum_{j=1}^{p} P(\beta_j) \right\} \tag{5}$$

where $\lambda > 0$ is a tuning parameter that controls the influence of the penalty term, and $P(\beta_j)$ is a function of the coefficients defining the penalty (Helwig, 2017b). The ridge penalty makes use of the ℓ_2 norm and is given $P(\beta_j) = \beta_j^2$, which is used in ridge regression (Hoerl & Kennard, 1970). Ridge regression shrinks predictor coefficients towards zero (but not to zero exactly) and is typically applied when predictors are highly correlated. Alternatively, the lasso penalty makes use of the ℓ_1 norm and is written $P(\beta_j) = |\beta_j|$, which is used in lasso regression (Tibshirani, 1996). Often applied in high-dimensional settings, lasso introduces sparsity by shrinking some coefficients to zero, creating a method for identifying useful predictors. However, it is known for arbitrarily selecting one predictor among a group of highly correlated predictors, which presents a limitation to data scenarios with highly correlated features. As a compromise, the elastic net penalty combines the ℓ_1 norm with the ℓ_2 norm, and can be written $P(\beta_j) = \alpha|\beta_j| + \frac{1}{2}(1-\alpha)\beta_j^2$, where $0 \leq \alpha \leq 1$ is a tuning parameter that designates a tradeoff between the two norms (Zou & Hastie, 2005). This tradeoff can be used to overcome lasso's arbitrary selection. Note that **cpfa** tunes α while `cv.glmnet` from **glmnet** tunes λ internally.

Support Vector Machine (SVM)

A support vector classifier finds an optimal hyperplane that separates two classes of points in a shared feature space (Boser et al., 1992; Cortes & Vapnik, 1995). One representation of the SVM objective is with a loss-penalty formulation. For n observations and p predictors, let $f(\mathbf{x}_i) = \beta_0 + \mathbf{h}(\mathbf{x}_i)^\top \boldsymbol{\beta}$, where $\mathbf{x}_i$ is the i-th observation's observed predictor vector for $i = 1, \ldots, n$, and where $\mathbf{h}(\mathbf{x}_i)^\top = (h_1(\mathbf{x}_i), h_2(\mathbf{x}_i), \ldots, h_p(\mathbf{x}_i))$ is a transformation of $\mathbf{x}_i$ into a higher (possibly infinite-dimensional) space via selected basis functions. A loss-penalty objective for SVMs can be stated as

$$\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{n} \sum_{i=1}^{n} (1 - y_i f(\mathbf{x}_i))_+ + \lambda \sum_{j=1}^{p} \|\beta_j\|_2^2, \tag{6}$$

where y_i is the i-th observation's binary response. Contained within $f(\mathbf{x}_i)$ are β_0 and $\boldsymbol{\beta}$, the model's intercept and slope coefficients, which are the parameters that define the separating hyperplane's location. Moreover, $\phi(r) = (1 - r)_+$ is hinge loss, where $(\cdot)_+$ indicates the positive part of a real-valued number, also written $\phi(r) = \max(1 - r, 0)$. In the second term, $\|\cdot\|_2^2$ is the ℓ_2 norm and λ is a tuning parameter that weights the penalty added to the loss (Zou, 2019). Hinge loss and its variants are the most frequently employed for SVMs and constitute one family of loss functions. We can re-express $f(\mathbf{x}_i)$ such that

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y_i \langle \mathbf{h}(\mathbf{x}_i), \mathbf{h}(\mathbf{x}) \rangle + \beta_0, \tag{7}$$

where α_i is a reparameterization of the slope coefficients, and where $\langle \mathbf{h}(\mathbf{x}_i), \mathbf{h}(\mathbf{x}) \rangle$ denotes the inner product between $\mathbf{h}(\mathbf{x}_i)$ and $\mathbf{h}(\mathbf{x})$ (Hastie et al., 2009, pg. 423). Rather than calculating this inner product, a kernel function can be used instead where $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{h}(\mathbf{x}_i), \mathbf{h}(\mathbf{x}_j) \rangle$. While the kernel function itself can be treated as a tuning parameter, the radial basis function kernel for SVMs is known to be useful and flexible (Pisner & Schnyer, 2020). The radial basis kernel function can be written $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$, where γ is taken as a tuning parameter. For SVMs with radial basis kernels, the complexity cost C is also important. Cost is a reparameterization of λ where $C = \frac{1}{\lambda}$. Note that **cpfa** only uses a radial basis kernel and tunes both C and γ.

Random Forest (RF)

Random forest (RF) takes the ideas of bagging and classification trees and extends them (see Haste et al., 2009 for a discussion of classification trees and bagging). For B identically distributed (but not independent) random variables, each with variance σ^2, the variance of their average can be given

$$\rho\sigma^2 + \frac{1 - \rho}{B}\sigma^2, \tag{8}$$

where ρ is their pairwise correlation (Hastie et al., 2009, pg. 588). Taking the random variables to be B bootstrap samples of some training set, bagging reduces the second term by increasing B. The addition of random forests is to randomly select a subset of predictors at each terminal node, for a given tree and bootstrap sample, splitting the tree and repeating many times over, and averaging the variance across subsets.

Choosing predictor subsets reduces ρ and thereby reduces the first term (Breiman, 2001). In practice, RF outperforms bagging and is one of several general classifiers that performs best on benchmark data sets, alongside boosting, SVMs, and NNs (Fernandez-Delgado et al., 2014). To optimize classification, three RF parameters are typically tuned with cross-validation, including: (1) the number of trees grown; (2) the number of predictors sampled at each node splitting; and (3) the minimum node size, where a larger minimum node size corresponds to smaller trees. Note that **cpfa** sets the number of predictors sampled at each node splitting to $\sqrt{R}$ (see Introduction) and tunes both number of trees and minimum node size.

Feed-forward Neural Networks (NN)

Neural networks (NNs) include a family of related statistical models that generalize linear regression by taking nonlinear functions of linear features (Ripley, 1994). A widely used NN is the feed-forward neural network (FFNN), also called a single layer perceptron, which solves a regression problem in two stages. In stage one, linear combinations of input features (called hidden units) are formed and nonlinearly transformed. In a second stage, linear combinations of these derived features are then taken and transformed using a separate function. Specifically, for $k = 1, \ldots, K$ classes, for $m = 1, \ldots M$ hidden units, and for p predictors, a FFNN classification model can be written

$$Y_k = g_k(\beta_{0_k} + \boldsymbol{\beta}_k^T \phi(\alpha_{0_m} + \boldsymbol{\alpha}_m^T X)), \tag{9}$$

where Y_k is the target response for the k-th class, and where X is a p-variate predictor vector (Hastie et al., 2009, pg. 392). Terms within ϕ represent the first stage, where α_{0_m} is the intercept for the m-th hidden unit, and where $\boldsymbol{\alpha}_m$ is a $p \times 1$ vector of coefficients weighing the p predictors for the m-th hidden unit. Frequently, the stage one transformation ϕ is the logistic function $\phi(r) = 1/(1 + e^{-r})$. Stage two is then given within g_k, where β_{0_k} is the intercept for the k-th class, and where $\boldsymbol{\beta}_k$ is a $M \times 1$ coefficient vector that weighs the derived features from stage one. For classification, the second stage function g_k is often the softmax function $g_k(r) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$, which is the transformation used in logistic multinomial regression (Bridle, 1990). Selecting a large M for a fixed g_k will often overfit the solution, so M is usually treated as a tuning parameter optimized by cross-validation (Hastie et al., 2009, pg. 393).

To estimate model parameters, FFNNs use either squared error or entropy (deviance). For squared error, the objective function is given

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - g_k(x_i))^2,$$

and for entropy/deviance, by

$$R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(g_k(x_i)),$$

both for $k = 1, \ldots, K$ classes and $i = 1, \ldots, N$ observations. Parameters are estimated using variants of gradient descent, also called back-propagation in NN literature. In order to reduce overfitting, R is typically regularized by adding a bias term with $R(\theta) + \lambda J(\theta)$, where $\lambda \geq 0$ weights the amount of added bias, and where

$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{k,m} \alpha_{km}^2.$$

This form of regularization is also called weight decay, and λ is usually treated as a tuning parameter to be optimized by cross-validation (Hastie et al., 2009, pgs. 395 - 397). Note that **cpfa** tunes both size and decay.

References

Boser, B. E., Guyon, I. M., & Vapnik, V. N. (1992, July). A training algorithm for optimal margin classifiers. In Proceedings of the fifth annual workshop on Computational learning theory (pp. 144-152).

Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5-32.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. Neurocomputing (pp. 227-236). Springer, Berlin, Heidelberg.

Cortes, C. & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273-297.

Cox, D. R. (1958). The regression analysis of binary sequences. Journal of the Royal Statistical Society: Series B (Methodological), 20(2), 215-232.

Faber, N. K. M., Bro, R., & Hopke, P. K. (2003). Recent developments in CANDECOMP/PARAFAC algorithms: a critical review. Chemometrics and Intelligent Laboratory Systems, 65(1), 119-137.

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems?. The Journal of Machine Learning Research, 15(1), 3133-3181.

Friedman, J. Hastie, T., & Tibshirani, R. (2010). Regularization Paths for Generalized Linear Models via Coordinate Descent. Journal of Statistical Software, 33(1), 1-22.

Gentle, J. E. (1998). Numerical linear algebra for applications in statistics. Springer Science & Business Media.

Harshman, R. A. (1970). Foundations of the PARAFAC procedure: Models and conditions for an" explanatory" multimodal factor analysis.

Harshman, R. A. & Lundy, M. E. (1994). PARAFAC: Parallel factor analysis. Computational Statistics and Data Analysis, 18, 39-72.

Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). The elements of statistical learning: data mining, inference, and prediction (Vol. 2, pp. 1-758). New York: springer.

Helwig, N. E. (2017a). Estimating latent trends in multivariate longitudinal data via Parafac2 with functional and structural constraints. Biometrical Journal, 59(4), 783-803.

Helwig, N. E. (2017b). Adding bias to reduce variance in psychological results: A tutorial on penalized regression. The Quantitative Methods for Psychology, 13(1), 1-19.

Helwig, N. E. (2019). multiway: Component Models for Multi-Way Data. R package version 1.0-6.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. Technometrics, 12(1), 55-67.

Kiers, H. A., Ten Berge, J. M., & Bro, R. (1999). PARAFAC2—Part I. A direct fitting algorithm for the PARAFAC2 model. Journal of Chemometrics: A Journal of the Chemometrics Society, 13(3-4), 275-294.

Liaw, A. & Wiener, M. (2002). Classification and Regression by randomForest. R News 2(3), 18–22.

Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., & Leisch, F. (2021). e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien. R package version 1.7-6.

Microsoft Corporation and Steve Weston (2022). doParallel: Foreach Parallel Adaptor for the 'parallel' Package. R package version 1.0.17.

Pisner, D. A., & Schnyer, D. M. (2020). Support vector machine. Machine Learning (pp. 101-121). Academic Press.

R Core Team (2022). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ripley, B. (1994). Neural networks and related methods for classification. Journal of the Royal Statistical Society: Series B (Methodological), 56(3), 409-437.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267-288.

Tomasi, G., & Bro, R. (2006). A comparison of algorithms for fitting the PARAFAC model. Computational Statistics & Data Analysis, 50(7), 1700-1734.

Venables, W. and Ripley, B. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0.

Zou, H. (2019). Classification with high dimensional features. Wiley Interdisciplinary Reviews: Computational Statistics, 11(1), e1453.

Zou, H. & Hastie, T. (2005). Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67(2), 301-320.

Acknowledgements

A special thanks to my mentor and advisor, Nathaniel E. Helwig, Ph.D., for his thoughtful advice on R package development.