Package ‘BaTFLED3D’

October 6, 2017

Title Bayesian Tensor Factorization Linked to External Data

Version 0.2.11

Description BaTFLED is a machine learning algorithm designed to make predictions and determine interactions in data that varies along three independent modes. For example BaTFLED was developed to predict the growth of cell lines when treated with drugs at different doses. The first mode corresponds to cell lines and incorporates predictors such as cell line genomics and growth conditions. The second mode corresponds to drugs and incorporates predictors indicating known targets and structural features. The third mode corresponds to dose and there are no dose-specific predictors (although the algorithm is capable of including predictors for the third mode if present). See 'BaTFLED3D_vignette.rmd' for a simulated example.

Depends R (>= 3.2.2)

License MIT + file LICENSE

LazyData true

RoxygenNote 5.0.1

Collate 'diagnostics.R' 'CP_model.R' 'diagonal.R' 'Tucker_model.R' 'exp_var.R' 'get_data_params.R' 'get_influence.R' 'get_model_params.R' 'im_2_mat.R' 'im_mat.R' 'input_data.R' 'kernelize.R' 'lower_bnd_Tucker.R' 'lower_bnd_CP.R' 'mk_model.R' 'mk_toy.R' 'mult_3d.R' 'nrmse.R' 'plot_preds.R' 'plot_roc.R' 'plot_test_RMSE.R' 'plot_test_cor.R' 'plot_test_exp_var.R' 'rmse.R' 'rot.R' 'safe_log.R' 'safe_prod.R' 'show_mat.R' 'test.R' 'test_CP.R' 'test_Tucker.R' 'test_results.R' 'train.R' 'train_CP.R' 'train_Tucker.R' 'update_core_Tucker.R' 'update_mode1_Tucker.R' 'update_mode2_Tucker.R' 'update_mode3_Tucker.R' 'update_mode1_CP.R' 'update_mode2_CP.R' 'update_mode3_CP.R'

Imports foreach, R6, iterators, rTensor, RColorBrewer

Suggests doMC, doParallel, knitr, rmarkdown, testthat

VignetteBuilder knitr

NeedsCompilation no

Author Nathan Lazar [aut, cre]

Maintainer Nathan Lazar <nathan.lazar@gmail.com>

Repository CRAN

Date/Publication 2017-10-06 20:26:57 UTC

R topics documented:

CP_model	*BaTFLED model object for 3-D response tensor with CP decomposition.*

Description

CP_model objects are 'R6' objects so that their values can be updated in place. The object is treated like an environment and components are accessed using the $ operator. When creating a new CP_model object it will be populated with default values and empty matrices. To initialize a CP_model call the initialize() method.

Usage

```
CP_model
```

Format

An R6Class generator object

Methods

new(data, params) Creates a new CP_model object with matrices sized accoring to the matrices in data.

rand_init(params) Initializes the CP_model with random values accoring to params.

See Also

get_model_params, input_data, Tucker_model

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)
```

diagonal — *Version of diag that has more consistent behavior*

Description

Version of diag that has more consistent behavior

Usage

```
diagonal(x, len = NA, ...)
```

Arguments

x	A vector, matrix or array with third mode length 1
len	numeric dimensions of new diagonal matrix to me made. Recycles values in x.
...	parameters passed to diag

Value

if x is a vector or integer, return a matrix with x on the diagonal. If x is a matrix, or degenerate array, return the diagonal of x.

Examples

```
diagonal(c(1,3))
diagonal(matrix(1:6, 2,3))
diagonal(5)
diagonal(c(5,2),3)
diagonal(array(1:12, dim=c(3,4,1)))
```

exp_var — *Get the explained variance for a set of predictions*

Description

Calculates 1-var(obs-pred)/var(obs). If verbose == TRUE the result is printed.

Usage

```
exp_var(obs, pred, verbose = F)
```

Arguments

obs	data.frame, vector or matrix
pred	data.frame, vector or matrix
verbose	logical indicating whether to print result

Value

numeric value of the explained variance

Examples

```
exp_var(rnorm(100) + seq(0,9.9,.1),  seq(0,9.9,.1))
```

---

get_data_params — *Get parameters for building a model with known relationships*

Description

Read in vector of arguments, check their types and add them to a list `params` for building a model of input and response data with known relationships. If a parameter isn't in the given list the default is used.

Usage

```
get_data_params(args)
```

Arguments

`args` A character vector of arguments (character strings) of the form "<name>=<value>". Values will be converted to logical or numeric when necessary. Accepted <names> are below. Defaults in parenthesis:

decomp Either 'CP' or 'Tucker'. (Tucker)

row.share Logical. Should the variance be shared across rows of the projection matrices? This will cause predictors to be or excluded for the whole model, instead of just for particular latent factors. (T)

seed Numeric. Seed used for random initialization. (NA)

scale Logical. Should the input data columns should be scaled to have mean 0 and standard deviation 1. (TRUE)

m1.rows Numeric. Number of rows (samples) for mode 1. (20)

m2.rows Numeric. Number of rows (samples) for mode 2. (25)

m3.rows Numeric. Number of rows (samples) for mode 3. (10)

m1.cols Numeric. Number of columns (predictors) for mode 1. (100)

m2.cols Numeric. Number of columns (predictors) for mode 2. (150)

m3.cols Numeric. Number of columns (predictors) for mode 3. (0)

R Numeric. If `decomp=='CP'` the dimension of the latent space for all modes. (4)

R1 Numeric. If `decomp=='Tucker'` the dimension of the core (latent space) for mode 1. (3)

R2 Numeric. If `decomp=='Tucker'` the dimension of the core (latent space) for mode 2. (3)

R3 Numeric. If `decomp=='Tucker'` the dimension of the core (latent space) for mode 3. (3)

A1.intercept Logical. Should a column of 1s be added to the input data for mode 1. (TRUE)

A2.intercept Logical. Should a column of 1s be added to the input data for mode 2. (TRUE)

A3.intercept Logical. Should a column of 1s be added to the input data for mode 3. (TRUE)

H1.intercept Logical. Should a column of 1s be added to the latent (H) matrix for mode 1. (TRUE)

H2.intercept Logical. Should a column of 1s be added to the latent (H) matrix for mode 2. (TRUE)

H3.intercept Logical. Should a column of 1s be added to the latent (H) matrix for mode 3. (TRUE)

m1.true Numeric. Number of predictors for mode 1 (not counting the constant) contributing to the response. (15)

m2.true Numeric. Number of predictors for mode 2 (not counting the constant) contributing to the response. (20)

m3.true Numeric. Number of predictors for mode 3 (not counting the constant) contributing to the response. (0)

A1.const.prob Numeric. Probability (0-1) of the constant term for mode 1 contributing to the response for mode 1. (1)

A2.const.prob Numeric. Probability (0-1) of the constant term for mode 2 contributing to the response. (1)

A3.const.prob Numeric. Probability (0-1) of the constant term for mode 3 contributing to the response. (1)

A.samp.sd Numeric. Standard deviation for sampling values for the projection (A) matrices. (1)

H.samp.sd Numeric. Standard deviation for sampling values for the latent (H) matrices. (1)

R.samp.sd Numeric. Standard deviation for sampling values for the core tensor. (1)

true.0D Numeric. 0 or 1, should a global intercept (0 dimensional intercept) be added to all responses? Only possible if `H1.intercept==H2.intercept==H3.intercept==TRUE`. `core.spar` is used if equal to `NA`. (NA)

true.1D.m[1-3] Numeric. Number of interactions of 1 dimension in the core tensor (non-zero elements on the edges of the core tensor if `H#.intercept==TRUE`). `core.spar` is used if equal to `NA`. (NA)

true.2D.m[1-3 m[1-3]] Numeric. Number of interactions of 2 dimensions in the core tensor (non-zero elements of the faces of the core tensor if `H#.intercept==TRUE`). `core.spar` is used if equal to `NA`. (NA)

true.3D Numeric. Number of interactions of 3 dimensions in the core tensor (non-zero elements internal to the core tensor). `core.spar` is used if equal to `NA`. (NA)

core.spar Numeric. Fraction of core elements that are non-zero. (1)

noise.sd Numeric. Relative standard deviation of noise added to response tensor. (0.1)

Value

list of parameters used by mk_toy function. Values in args that are not accepted parameters will be excluded and a warning displayed.

See Also

mk_toy

Examples

```
args <- c('decomp=Tucker', 'row.share=F',
          'A1.intercept=T', 'A2.intercept=T', 'A3.intercept=F',
          'H1.intercept=T', 'H2.intercept=T', 'H3.intercept=T',
          'R1=4', 'R2=4', 'R3=2')
data.params <- get_data_params(args)
```

---

get_influence — *Given a* model *object, rank the input predictors (and combinations thereof) by thier influence on the output*

Description

If method is 'add' then the baseline prediction is made using just the constant coefficients (if used) and the mean squared error (MSE) is measured between the baseline and predictions made with each predictor added alone (univariate analysis).

Usage

```
get_influence(m, d, method = "sub", interactions = TRUE)
```

Arguments

m	Tucker_model or CP_model object
d	input_data object
method	string 'sub' or 'add' indicating whether to start with a full or empty feature vector and remove or add features to judge their influence.
interactions	logical indicating whether to get influence for two-way interactions between predictors (def: sub)

Details

If method is 'sub' then the baseline is made using all predictors and MSE measured for predictions made with each predictor removed.

If interactions==TRUE then MSE for predictions made with predictors for each mode interacting are measured `get_model_params` *Get parameters to build a BaTFLED model*

Description

Read in vector of arguments, check their types and add them to a list `params` for model training. If a parameter isn't in the given list the default is used.

Usage

```
get_model_params(args)
```

Arguments

`args` A character vector of arguments (character strings) of the form "<name>=<value>". Values will be converted to logical or numeric when possible. Accepted <names> are below. Defaults in parenthesis:

- **decomp** Either 'CP' or 'Tucker'. (Tucker)
- **row.share** Logical. Should the variance be shared across rows of the projection matrices? This will cause predictors to be or excluded for the whole model, instead of just for particular latent factors. (F)
- **seed** Numeric. Seed used for random initialization. (NA)
- **verbose** Logical. Display more messages during training. (F)
- **parallel** Logical. Perform operations in parallel when possible. (T)
- **cores** Numeric. The number of parallel threads to use. (2)
- **lower.bnd** Logical. Should the lower bound be calculated during training. Setting to FALSE saves time (F)
- **RMSE** Logical. Should the root mean squared error for the training data be calculated during training. (T)
- **cor** Logical. Should the Pearson correlation for the training data be calculated during training. (T)
- **A1.intercept** Logical. Add a constant column to the mode 1 predictors. (T)
- **A2.intercept** Logical. Add a constant column to the mode 2 predictors. (T)
- **A3.intercept** Logical. Add a constant column to the mode 3 predictors. (F)
- **H1.intercept** Logical. Add a constant column to the mode 1 latent (H) matrix. (F)
- **H2.intercept** Logical. Add a constant column to the mode 2 latent (H) matrix. (F)
- **H3.intercept** Logical. Add a constant column to the mode 3 latent (H) matrix. (F)
- **R** Numeric. Number of latent factors used in a CP model. (3)
- **R1** Numeric. Number of latent factors used for mode 1 in a Tucker decomposition. (3)

R2 Numeric. Number of latent factors used for mode 2 in a Tucker decomposition. (3)

R3 Numeric. Number of latent factors used for mode 3 in a Tucker decomposition. (3)

core.updates Numeric. Number of core elements to update each round for stochastic training. (all)

m1.alpha Numeric. Prior for the 'shape' parameter of the gamma distribution on the precision values in the mode 1 projection (A) matrix. Set this to a small value (ex. 1e-10) to encourage sparsity in mode 1 predictors. (1)

m2.alpha Numeric. Same as above for mode 2. (1)

m3.alpha Numeric. Same as above for mode 3. (1)

m1.beta Numeric. Prior for the 'scale' parameter of the gamma distribution on the precision values in the mode 1 projection (A) matrix. Set this to a large value (ex. 1e10) to encourage sparsity in mode 1 predictors. Note this should stay balanced with m1.alpha so thir product is 1. (1)

m2.beta Numeric. Same as above for mode 2. (1)

m3.beta Numeric. Same as above for mode 3. (1)

A.samp.sd Numeric. Standard deviation used when initializing values in the projection (A) matrices. (1)

H.samp.sd Numeric. Standard deviation used when initializing values in the latent (H) matrices. (1)

R.samp.sd Numeric. Standard deviation used when initializing values in the core tensor for Tucker models. (1)

A.var Numeric. Initial variance for projection (A) matrices. (1)

H.var Numeric. Initial variance for latent (H) matrices. (1)

R.var Numeric. Initial variance for the core tensor in Tucker models. (1)

random.H Logical. Should the latent matrices be initialized randomly or be the result of multiplying the input data by the projection matrices. (T)

core.0D.alpha Numeric. Prior for the 'scale' parameter of the gamma distribution on the precision value in the element of the core tensor corresponding to the intercept for all three modes (core.mean[1,1,1]). Only used for Tucker models when all H intercepts are true. Set this to a small value (ex. 1e-10) to encourage sparsity in core predictor. (1)

core.1D.alpha Numeric. As above for values corresponding to the intercepts for two modes (core.mean[1,1,], core.mean[1,,1] and core.mean[,1,1]). (1)

core.2D.alpha Numeric. As above for values corresponding to the intercepts for one mode (core.mean[1,,], core.mean[,1,] and core.mean[,,1]). (1)

core.3D.alpha Numeric. As above for values not corresponding to intercepts. (1)

core.0D.beta Numeric. As above but a prior for the 'scale' parameter. (1)

core.1D.beta Numeric. As above but a prior for the 'scale' parameter. (1)

core.2D.beta Numeric. As above but a prior for the 'scale' parameter. (1)

core.3D.beta Numeric. As above but a prior for the 'scale' parameter. (1)

m1.sigma2 Numeric. Variance for the mode 1 latent (H) matrix. Set small to link the values in the latent matrices to the product of the input and projection matrices. If there is no input data, set to one or larger. (0.01)

m2.sigma2 Numeric. As above for mode 2. (0.01)

m3.sigma2 Numeric. As above for mode 3. (0.01)

sigma2 Numeric. Variance parameter for the response tensor or 'auto' (default). If set to 'auto' then the square-root of the variance of the training responses is used.

remove.start Numeric. The iteration to begin removing predictors if any of `m1.remove.lmt`, `m2.remove.lmt`, `m3.remove.lmt` or `remove.per` are set. (Inf)

remove.per Numeric. Percentage of predictors to remove with the lowest mean of squared values across rows of the projection matrix. (0)

m1.remove.lmt Numeric. Remove a mode 1 predictor if the mean squared value of its row in the projection matrix drop below this value. (0)

m2.remove.lmt As above for mode 2. (0)

m3.remove.lmt As above for mode 3. (0)

early.stop Numeric. Stop training if the lower bound value changes by less than this value. (0)

plot Logical. Show plots while training

show.mode Numeric vector. Display images of the projection and latent matrices for these modes while training. (c(1,2,3))

update.order Numeric vector. Update the modes in this order (c(3,2,1))

Value

list of parameters used by `train` function. Values in `args` that are not model parameters will be excluded and a warning displayed.

See Also

`CP_model` `Tucker_model`

Examples

```
args <- c('decomp=Tucker', 'row.share=F',
          'A1.intercept=T', 'A2.intercept=T', 'A3.intercept=F',
          'H1.intercept=T', 'H2.intercept=T', 'H3.intercept=T',
          'plot=T', 'verbose=F','R1=4', 'R2=4', 'R3=2',
          'm1.alpha=1e-10', 'm2.alpha=1e-10', 'm3.alpha=1',
          'm1.beta=1e10', 'm2.beta=1e10', 'm3.beta=1',
          'core.3D.alpha=1e-10', 'core.3D.beta=1e10',
          'parallel=T', 'cores=5', 'lower.bnd=T',
          'update.order=c(3,2,1)', 'show.mode=c(1,2,3)',
          'wrong=1')
model.params <- get_model_params(args)
```

im_2_mat *Plot heatmaps of two matrices in red and blue*

Description

Displays two heatmaps of matrices using red and blue colors. Options to scale and sort as well as any other graphical parameters with ... Sorting attempts to match columns between the two matrices using their correlation over rows. If `sort==TRUE` then the new ordering for the second matrix is returned.

Usage

```
im_2_mat(x1, x2, high = "red", xaxt = "n", yaxt = "n", scale = "col",
  absol = FALSE, sort = TRUE, center = FALSE, main1 = "", main2 = "",
  ...)
```

Arguments

`x1`	matrix
`x2`	matrix
`high`	string of either 'red' or 'blue' used to show higher values
`xaxt`	string indicating how to display the x axis. Suppress x axis with 'n'
`yaxt`	string indicating how to display the y axis. Suppress y axis with 'n'
`scale`	logical indicating whether the matrices should be z scaled to have columns with norm zero and standard deviation one.
`absol`	logical indicating whether to take absolute value of the entries before plotting
`sort`	logical indicating whether the columns of the matrix should be sorted in decreasing order of their means
`center`	logical indicating wether to center ranges for x and y around zero
`main1`	string to be used as the main title for the first matrix image
`main2`	string to be used as the main title for the second matrix image
`...`	other graphical parameters passed to image

Value

If `sort==TRUE` the ordering of the second matrix used to match columns.

Examples

```
par(mfrow=c(1,2))
im_2_mat(matrix(1:12, nrow=3, ncol=4),  matrix(13:24, nrow=3, ncol=4), sort=FALSE, scale=FALSE)
im_2_mat(matrix(1:12, nrow=3, ncol=4),  matrix(13:24, nrow=3, ncol=4), sort=TRUE, scale=FALSE)
im_2_mat(matrix(1:12, nrow=3, ncol=4),  matrix(13:24, nrow=3, ncol=4), sort=TRUE, scale=TRUE)
im_2_mat(matrix(1:12, nrow=3, ncol=4),  matrix(13:24, nrow=3, ncol=4), sort=FALSE,
        scale=FALSE, center=TRUE)
```

im_mat — *Plot a heatmap of a matrix in red and blue*

Description

Displays a heatmap of a matrix using red and blue colors. Options to scale and sort as well as any other graphical parameters with ...

Usage

```
im_mat(x, high = "red", xaxt = "n", yaxt = "n", sort = FALSE,
  scale = FALSE, ballance = FALSE, zlim = NA, ...)
```

Arguments

`x`	matrix
`high`	string of either 'red' or 'blue' used to show higher values
`xaxt`	string indicating how to display the x axis. Suppress x axis with 'n'
`yaxt`	string indicating how to display the y axis. Suppress y axis with 'n'
`sort`	logical indicating whether the columns of the matrix should be sorted in decreasing order of their means
`scale`	logical indicating whether the matrix should be z scaled to have columns with norm zero and standard deviation one.
`ballance`	logical indicating whether to expand the range so it stays centered at zero
`zlim`	numeric bounds on the max and min range for colors.
`...`	other graphical parameters passed to image

Value

none

Examples

```
im_mat(matrix(1:12, nrow=3, ncol=4), sort=FALSE, scale=FALSE)
im_mat(matrix(1:12, nrow=3, ncol=4), sort=TRUE, scale=FALSE)
im_mat(matrix(1:12, nrow=3, ncol=4), sort=FALSE, scale=TRUE)
im_mat(matrix(1:12, nrow=3, ncol=4), sort=TRUE, scale=TRUE)
```

`input_data` *Object storing input data for BaTFLED algorithm with 3-D response tensor.*

Description

Object storing input data for BaTFLED algorithm with 3-D response tensor.

Usage

```
input_data
```

Format

An object of class `R6ClassGenerator` of length 24.

Slots

`mode1.X` matrix of predictors for mode 1

`mode2.X` matrix of predictors for mode 2

`mode3.X` matrix of predictors for mode 3

`resp` three dimensional array of responses with dimensions matching the number of rows in mode1.X, mode2.X and mode3.X

Examples

```
a <- input_data$new(mode1.X = matrix(rnorm(30), nrow=3, ncol=10),
                    mode2.X = matrix(rnorm(36), nrow=4, ncol=9),
                    mode3.X = matrix(rnorm(40), nrow=5, ncol=8),
                    resp = array(rnorm(60), dim=c(3,4,5)))
im_mat(a$mode1.X)
im_mat(a$mode2.X)
im_mat(a$mode3.X)
im_mat(a$resp[,,1])
```

`kernelize` *Transform a matrix of input data into a matrix of kernel simmilarities values*

Description

The input matrices should have samples as the rows and features as columns. A kernel will computed across all samples in the first matrix with respect to the samples in the second matrix. The two matrices must have the same features. If all features are binary 0,1, then the Jaccard similarity kernel will be used, otherwise, a Gaussian kernel with standard deviation equal to s times the mean euclidean distances between samples in the second matrix. If there are samples with all NA values, they will not appear in the kernel matrix columns. The row for that sample will just be all NAs.

Usage

```
kernelize(m1, m2 = NA, s = 1)
```

Arguments

m1	matrix on samples X features to compute kernels on
m2	matrix of samples X features to compute kernels with respect to.
s	numeric multiplier of standard deviation for the Gaussian kernels (default:1).

Value

matrix of similarities between rows of m1 and rows of m2.

Examples

```
m1 <- matrix(rnorm(200), 8, 25, dimnames=list(paste0('sample.', 1:8), paste0('feat.', 1:25)))
m2 <- matrix(rnorm(100), 4, 25, dimnames=list(paste0('sample.', 9:12), paste0('feat.', 1:25)))
kernelize(m1, m1)
kernelize(m1, m1, s=.5)
kernelize(m2, m1)
m1 <- matrix(rbinom(200, 1, .5), 8, 25,
             dimnames=list(paste0('sample.', 1:8), paste0('feat.', 1:25)))
m2 <- matrix(rbinom(25, 1, .5), 1, 25,
             dimnames=list(c('sample.9'), paste0('feat.', 1:25)))
kernelize(m1, m1)
kernelize(m2, m1)
```

---

lower_bnd_CP *Calculate the lower bound of the log likelihood for a trained CP model*

Description

Calculate the lower bound of the log likelihood for a trained CP model

Usage

```
lower_bnd_CP(m, d)
```

Arguments

m	object of the class CP_model
d	object of the class input_data

Value

Returns a numerical value (should be negative)

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

train(d=train.data, m=toy.model, new.iter=1, params=model.params)

lower_bnd_CP(toy.model, train.data)
```

---

lower_bnd_Tucker — *Calculate the lower bound of the log likelihood for a trained Tucker model*

Description

Calculate the lower bound of the log likelihood for a trained Tucker model

Usage

```
lower_bnd_Tucker(m, d)
```

Arguments

m	object of the class `Tucker_model`
d	object of the class `input_data`

Value

Returns a numerical value (should be negative)

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

lower_bnd_Tucker(toy.model, train.data)
```

mk_model *Make a new model object*

Description

This function is a wrapper calling Tucker_model$new() or CP_model$new() depending on whether `params$decomp=='Tucker'` or `params$decomp=='CP'`

Usage

```
mk_model(d, params)
```

Arguments

d	An `input_data` object. See `input_data`.
params	A list of parameter values `get_model_params`.

Value

`CP_model` or `Tucker_model` object

See Also

Tucker_model, CP_model

mk_toy *Make a toy dataset to test the 3d BaTFLED model.*

Description

Returns a toy model with the specified size, sparsity and noise generated either with a CP or Tucker factorization model. Values in predictor matrices (X1, X2, X3) are pulled from a standard normal distribuion. Dummy names are given to the predictors.

Usage

```
mk_toy(params)
```

Arguments

params	list of parameters created with `get_data_params`

Value

a list containing elements of the model

mode1.X Input data for mode 1

mode2.X Input data for mode 2

mode3.X Input data for mode 3

mode1.A Projection matrix for mode 1

mode2.A Projection matrix for mode 2

mode3.A Projection matrix for mode 3

mode1.H Latent matrix for mode 1

mode2.H Latent matrix for mode 2

mode3.H Latent matrix for mode 3

core Core tensor if `params$decomp=='Tucker'`

resp Response tensor

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)

data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
```

---

mult_3d — *Multiply three matrices (or vectors) through a given core tensor to form a three dimensional tensor.*

Description

The package 'rTensor' is required and the number of columns of x, y and z must match the dimensions of core.

Usage

```
mult_3d(core, x, y, z, names = T)
```

Arguments

`core`	array
`x`	matrix to multiply by the first mode of `core`
`y`	matrix to multiply by the second mode of `core`
`z`	matrix to multiply by the third mode of `core`
`names`	logical indicating whether to keep the dimension names

Value

Array with sizes given by the number of rows in x, y and z

Examples

```
mult_3d(array(1:24, dim=c(2,3,4)), matrix(1:4,2,2), matrix(1:6,2,3), matrix(1:8,2,4))
```

---

nrmse *Computes the normalized root mean squared error*

Description

Computes the normalized root mean squared error

Usage

```
nrmse(obs, pred)
```

Arguments

obs	observed vector, matrix or data.frame
pred	predicted vector, matrix or data.frame

Value

numeric value of the root mean squared error normalized to the standard deviation of the observed data

---

plot_preds *Make a scatterplot of observed vs. predicted values*

Description

If there are more than 25,0000 points then they are subsampled down to 25,000. Observed values are on the x axis predicted values on the y. A blue line shows the diagonal. Points are transparent to show dense clusters. Predictions for points where the true value is not known are plotted at zero in blue.

Usage

```
plot_preds(pred, true, show.na = T, ...)
```

Arguments

- `pred` matrix or vector of predicted values
- `true` matrix or vector of predicted values
- `show.na` logical, display NA values as blue dots at the mean for the x or y axis (def: T)
- `...` other parameters passed to plot

Value

none

Examples

```
x <- seq(-10,10, 0.01)+rnorm(2001)
y <- seq(-10,10, 0.01)+rnorm(2001)
x[sample(2001, 100)] <- NA
plot_preds(y, x)
```

---

plot_roc — *Plot reciever operating characteristic (ROC) curves for two projection (A) matrices*

Description

This is a little different than a typical ROC curve since any rows of the true matrix that are non-zero are treated as equal true positives.

Usage

```
plot_roc(true, pred, main = character(0))
```

Arguments

- `true` projection matrix where rows of true predictors have non-zero values
- `pred` projection matrix where rows of learned predictors have larger values
- `main` title of the ROC plot

plot_test_cor *Plot correlation results from test data*

Description

Plot correlation results from test data

Usage

```
plot_test_cor(test.results, ylim = "default", main = NA,
  method = "pearson", baselines = c(warm = NA, m1 = NA, m2 = NA, m3 = NA,
  m1m2 = NA, m1m3 = NA, m2m3 = NA, m1m2m3 = NA))
```

Arguments

test.results	results generated with test_results
ylim	limits for the y-axis (NA)
main	Main title of the plot
method	Either 'pearson' or 'spearman' correlations
baselines	named vector of baseline values to draw as dotted horizontal lines e.g. c('warm'=0, 'm1'=0, 'm1m2'=0, 'm1m2m3'=0)

plot_test_exp_var *Plot explained variance results from test data*

Description

Plot explained variance results from test data

Usage

```
plot_test_exp_var(test.results, ylim = "default", main = NA,
  baselines = c(warm = NA, m1 = NA, m2 = NA, m3 = NA, m1m2 = NA, m1m3 = NA,
  m2m3 = NA, m1m2m3 = NA))
```

Arguments

test.results	an object generated with test_results
ylim	Limits of the y-axis.
main	Main title of the plot
baselines	named vector of baseline values to draw as dotted horizontal lines e.g. c('warm'=0, 'm1'=0, 'm1m2'=0, 'm1m2m3'=0)

plot_test_RMSE — *Plot RMSE results from test data*

Description

Plot RMSE results from test data

Usage

```
plot_test_RMSE(test.results, ylim = "default", main = "Test RMSEs",
  baselines = c(warm = NA, m1 = NA, m2 = NA, m3 = NA, m1m2 = NA, m1m3 = NA,
  m2m3 = NA, m1m2m3 = NA))
```

Arguments

`test.results`	An object created with `test_results`
`ylim`	Limits of the y-axis (default is (0, 1.5))
`main`	Main title of the plot
`baselines`	named vector of baseline values to draw as dotted horizontal lines e.g. c('warm'=0, 'm1'=0, 'm1m2'=0, 'm1m2m3'=0)

rmse — *Updates the root mean squared error for training data. Predicting both from data and from just the latent (H) matrices.*

Description

Updates the root mean squared error for training data. Predicting both from data and from just the latent (H) matrices.

Usage

```
rmse(m, d, verbose = T)
```

Arguments

`m`	training object
`d`	data object
`verbose`	Logical indicating whether to print the results (TRUE)

Value

numeric value of the explained variance

rot *Rotate a matrix for printing*

Description

Rotates a matrix so that when view is called the rows and columns appear in the same order as when looking at the matrix with print

Usage

```
rot(m)
```

Arguments

m	matrix

Value

matrix that has been transposed and the columns reversed

Examples

```
# Normally image shows a matrix with the first entry in the bottom left
# With rot the image is shown in the same order as print
```

safe_log *Take logarithm avoiding underflow*

Description

Returns the normal log if there is no underflow. If there is underflow, then returns the minimum for which log can return (-744.4401)

Usage

```
safe_log(x)
```

Arguments

x	vector

Value

vector log in base e of input or minimum possible log value of -744.4401

Examples

```
log(c(1e-323, 1e-324))      # gives -Inf for the second value
safe_log(c(1e-323, 1e-324)) # gives the minimum value of -744.4401
```

`safe_prod` *Takes the product of two matrices adding a column of constants if necessary to the first matrix.*

Description

Takes the product of two matrices adding a column of constants if necessary to the first matrix.

Usage

```
safe_prod(A, B)
```

Arguments

A	matrix one
B	matrix two

Value

matrix product of A and B

`show_mat` *Plot matrices from a model object with im_mat*

Description

Plot matrices from a model object with im_mat

Usage

```
show_mat(m, d, show.mode, scale = F)
```

Arguments

m	model object created with mk_model
d	input data object created with get_input_data
show.mode	vector of modes whose projection and latent matrices are to be displayed
scale	Logical should the columns of matrices be scaled

`test` *Get test predictions for a 3D BaTFLED model.*

Description

This is just a wrapper that calls test_CP or test_Tucker depending on the type of model provided.

Usage

```
test(d, m, ...)
```

Arguments

`d`	object of the class `input_data` created with input_data()
`m`	object of the class `CP_model` or `Tucker_model` created with mk_model()
`...`	extra parameters passed to test_CP or test_Tucker

Value

An array of predicted responses the same size as `m$resp`.

`test_CP` *Perform 'cold start' prediction using BaTFLED algorthm for CP models*

Description

Perform 'cold start' prediction using BaTFLED algorthm for CP models

Usage

```
test_CP(d, m)
```

Arguments

`d`	an input data object created with `input_data`
`m`	a `CP_model` object created with `mk_model`

Value

Response tensor generated by multiplying the input data through the trained model

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

train(d=train.data, m=toy.model, new.iter=1, params=model.params)

resp <- test_CP(train.data, toy.model)
```

---

test_results   *Get RMSE & explained variance for warm and cold test results*

Description

Get RMSE & explained variance for warm and cold test results

Usage

```
test_results(m, d, test.results = numeric(0), verbose = T,
  warm.resp = numeric(0), test.m1 = numeric(0), test.m2 = numeric(0),
  test.m3 = numeric(0), test.m1m2 = numeric(0), test.m1m3 = numeric(0),
  test.m2m3 = numeric(0), test.m1m2m3 = numeric(0))
```

Arguments

m	a CP_model or Tucker_model object
d	an input data object created with input_data
test.results	an object generated by this function that will combined with the new results
verbose	Logical indicating whether to print the resulting prediction measures (TRUE)
warm.resp	True responses for warm test data (optional).
test.m1	True responses for mode 1 cold test data (optional).
test.m2	True responses for mode 2 cold test data (optional).
test.m3	True responses for mode 3 cold test data (optional).
test.m1m2	True responses for mode 1/2 combination cold test data (optional).
test.m1m3	True responses for mode 1/3 combination cold test data (optional).
test.m2m3	True responses for mode 2/3 combination cold test data (optional).
test.m1m2m3	True responses for mode 1/2/3 combination cold test data (optional).

Value

list of results TODO: add more here

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)

# Make training data object excluding the first two samples for modes 1 & 2.
train.data <- input_data$new(mode1.X=toy$mode1.X[-(1:2),-1],
                             mode2.X=toy$mode2.X[-(1:2),-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
# Remove some responses for warm prediction
warm.ind <- sample(1:prod(dim(train.data$resp)), 20)
warm.resp <- train.data$resp[warm.ind]
train.data$resp[warm.ind] <- NA

# Make testing objects
m1.test.data <- input_data$new(mode1.X=toy$mode1.X[1:2,-1],
                               mode2.X=toy$mode2.X[-(1:2),-1],
                               mode3.X=toy$mode3.X[,-1],
                               resp=toy$resp[1:2,-(1:2),])
m2.test.data <- input_data$new(mode1.X=toy$mode1.X[-(1:2),-1],
                               mode2.X=toy$mode2.X[1:2,-1],
                               mode3.X=toy$mode3.X[,-1],
                               resp=toy$resp[-(1:2),1:2,])
m1m2.test.data <- input_data$new(mode1.X=toy$mode1.X[1:2,-1],
                                 mode2.X=toy$mode2.X[1:2,-1],
                                 mode3.X=toy$mode3.X[,-1],
                                 resp=toy$resp[1:2,1:2,])

model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)
toy.model$iter <- 1

test.results <- numeric(0)
test_results(m=toy.model, d=train.data, warm.resp=warm.resp,
             test.m1=m1.test.data, test.m2=m2.test.data,
             test.m1m2=m1m2.test.data)
```

---

test_Tucker   *Perform 'cold start' prediction for Tucker models*

Description

Perform 'cold start' prediction for Tucker models

Usage

```
test_Tucker(d, m)
```

Arguments

`d`	an input data object created with `input_data`
`m`	a `Tucker_model` object created with `mk_model`

Value

Response tensor generated by multiplying the input data through the trained model

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

resp <- test_Tucker(train.data, toy.model)
```

`train` *Train model using BaTFLED algorthm*

Description

Model objects are updated in place to avoid memory issues. Nothing is returned.

Usage

```
train(d, m, ...)
```

Arguments

`d`	an input data object created with `input_data`
`m`	a `CP_model` or `Tucker_model` object created with `mk_model`
`...`	extra arguments (params) passed to train_CP or train_Tucker

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

train(d=train.data, m=toy.model, new.iter=1, params=model.params)
```

---

train_CP *Train a CP model.*

Description

Model objects are updated in place to avoid memory issues. Nothing is returned.

Usage

```
train_CP(d, m, new.iter = 1, params)
```

Arguments

`d`	an input data object created with `input_data`
`m`	a `CP_model` object created with `mk_model`
`new.iter`	numeric number of iterations to run (def: 1)
`params`	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

train(d=train.data, m=toy.model, new.iter=1, params=model.params)
```

train_Tucker — *Train a Tucker model using BaTFLED algorthm*

Description

Model objects are updated in place to avoid memory issues. Nothing is returned.

Usage

```
train_Tucker(d, m, new.iter = 1, params)
```

Arguments

d	an input data object created with input_data
m	a Tucker_model object created with mk_model
new.iter	numeric number of iterations to run (def: 1)
params	List of parameters created with get_model_params()

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

train(d=train.data, m=toy.model, new.iter=1, params=model.params)
```

Tucker_model — *Factorization object for 3D Tucker models.*

Description

Tucker_model objects are 'R6' objects so that their values can be updated in place. The object is treated like an environment and components are accessed using the $ operator. When creating a new Tucker_model object it will be populated with default values and empty matrices. To initialize a Tucker_model call the initialize() method.

Usage

```
Tucker_model
```

Format

An R6Class generator object

Members

iter Integer showing the number of iterations that have been run on this object.

early.stop Stop if the lower bound increases by less than this value.

lower.bnd Vector storing the lower bound values during training.

RMSE Vector of the root mean squared error of the predictions during training.

H.RMSE vector of the root mean squared error of predictions made by multiplying the H matrices.

exp.var Vector of the explained variance of predictions during training.

p.cor Vector of the Pearson correlation of predictions during training.

s.cor Vector of the Spearman correlation of predictions during training.

times Vector of the time taken for each update iteration.

core.mean Mean parameters of the q Gaussian distributions in the core tensor.

core.var Variance parameters of the q Gaussian distributions in the core tensor.

core.lambda.shape Prior for the shape parameter of the gamma distribution on the core precision.

core.lambda.scale Prior for the scale parameter of the gamma distribution on the core precision.

resp array storing the predicted response tensor.

delta binary array indicating whether the response is observed.

core.var variance parameters of the q Gaussian distributions in the core tensor.

m1Xm1X Product of mode1.X with itself stored to avoid recalculating.

m2Xm2X Product of mode2.X with itself stored to avoid recalculating.

m3Xm3X Product of mode3.X with itself stored to avoid recalculating.

mode1.lambda.shape Matrix storing the shape parameters for the gamma distributions on the mode 1 projection (A) matrix.

mode1.lambda.scale Matrix storing the scale parameters for the gamma distributions on the mode 1 projection (A) matrix.

mode2.lambda.shape Matrix storing the shape parameters for the gamma distributions on the mode 2 projection (A) matrix.

mode2.lambda.scale Matrix storing the scale parameters for the gamma distributions on the mode 2 projection (A) matrix.

mode3.lambda.shape Matrix storing the shape parameters for the gamma distributions on the mode 3 projection (A) matrix.

mode3.lambda.scale Matrix storing the scale parameters for the gamma distributions on the mode 3 projection (A) matrix.

mode1.A.mean Matrix storing the mean parameters for the normal distributions on the mode 1 projection (A) matrix.

mode1.A.cov Array storing the covariance parameters for the normal distributions on the mode 1 projection (A) matrix.

mode2.A.mean Matrix storing the mean parameters for the normal distributions on the mode 2 projection (A) matrix.

mode2.A.cov Array storing the covariance parameters for the normal distributions on the mode 2 projection (A) matrix.

mode3.A.mean Matrix storing the mean parameters for the normal distributions on the mode 3 projection (A) matrix.

mode3.A.cov Array storing the covariance parameters for the normal distributions on the mode 3 projection (A) matrix.

mode1.H.mean Matrix storing the mean parameters for the normal distributions on the mode 1 latent (H) matrix.

mode1.H.var Matrix storing the variance parameters for the normal distributions on the mode 1 latent (H) matrix.

mode2.H.mean Matrix storing the mean parameters for the normal distributions on the mode 2 latent (H) matrix.

mode2.H.var Matrix storing the variance parameters for the normal distributions on the mode 2 latent (H) matrix.

mode3.H.mean Matrix storing the mean parameters for the normal distributions on the mode 3 latent (H) matrix.

mode3.H.var Matrix storing the variance parameters for the normal distributions on the mode 3 latent (H) matrix.

sigma2 Variance for the response tensor.

m1.sigma Variance for the mode 1 latent (H) matrix.

m2.sigma Variance for the mode 2 latent (H) matrix.

m3.sigma Variance for the mode 3 latent (H) matrix.

m1.alpha Prior shape parameter for the gamma distribution on the precision of the mode 1 projection (A) matrix.

m1.beta Prior scale paramet for the gamma distribution on the precision of the mode 1 projection (A) matrix.

m2.alpha Prior shape parameter for the gamma distribution on the precision of the mode 2 projection (A) matrix.

m2.beta Prior scale paramet for the gamma distribution on the precision of the mode 2 projection (A) matrix.

m3.alpha Prior shape parameter for the gamma distribution on the precision of the mode 3 projection (A) matrix.

m3.beta Prior scale paramet for the gamma distribution on the precision of the mode 3 projection (A) matrix.

core.alpha Prior shape parameter for the gamma distribution on the precision of the core tensor.

core.beta Prior scale parameter for the gamma distribution on the precision of the core tensor.

core.0D.alpha Prior shape parameter for the gamma distribution on the precision of the 0D subset of the core tensor.

core.0D.beta Prior scale parameter for the gamma distribution on the precision of the 0D subset of the core tensor.

core.1D.alpha Prior shape parameter for the gamma distribution on the precision of the 1D subset of the core tensor.

core.1D.beta Prior scale parameter for the gamma distribution on the precision of the 1D subset of the core tensor.

core.2D.alpha Prior shape parameter for the gamma distribution on the precision of the 2D subset of the core tensor.

core.2D.beta Prior scale parameter for the gamma distribution on the precision of the 2D subset of the core tensor.

core.3D.alpha Prior shape parameter for the gamma distribution on the precision of the 3D subset of the core tensor.

core.3D.beta Prior scale parameter for the gamma distribution on the precision of the 3D subset of the core tensor.

Methods

`new(data, params)` Creates a new `Tucker_model` object with matrices sized accoring to the matrices in `data`.

`rand_init(params)` Initializes the `Tucker_model` with random values accoring to `params`.

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X[,-1],
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)
```

`update_core_Tucker` *Update values in the core tensor for a Tucker model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_core_Tucker(m, d, params)
```

Arguments

m	A `Tucker_model` object created with `mk_model`
d	Input data object created with `input_data`
params	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_core_Tucker(m=toy.model, d=train.data, params=model.params)
```

---

update_mode1_CP   *Update the first mode in a CP model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode1_CP(m, d, params)
```

Arguments

m	A `CP_model` object created with `mk_model`
d	Input data object created with `input_data`
params	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode1_CP(m=toy.model, d=train.data, params=model.params)
```

update_mode1_Tucker *Update the first mode in a Tucker model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode1_Tucker(m, d, params)
```

Arguments

m	A `Tucker_model` object created with `mk_model`
d	Input data object created with `input_data`
params	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode1_Tucker(m=toy.model, d=train.data, params=model.params)
```

update_mode2_CP *Update the second mode in a CP model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode2_CP(m, d, params)
```

Arguments

m	A `CP_model` object created with `mk_model`
d	Input data object created with `input_data`
params	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode2_CP(m=toy.model, d=train.data, params=model.params)
```

---

update_mode2_Tucker *Update the second mode in a Tucker model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode2_Tucker(m, d, params)
```

Arguments

m	A `Tucker_model` object created with `mk_model`
d	Input data object created with `input_data`
params	List of parameters created with `get_model_params()`

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode2_Tucker(m=toy.model, d=train.data, params=model.params)
```

update_mode3_CP *Update the third mode in a CP model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode3_CP(m, d, params)
```

Arguments

m	A CP_model object created with mk_model
d	Input data object created with input_data
params	List of parameters created with get_model_params()

Examples

```
data.params <- get_data_params(c('decomp=CP'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=CP'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode3_CP(m=toy.model, d=train.data, params=model.params)
```

update_mode3_Tucker *Update the third mode in a Tucker model.*

Description

Update is performed in place to avoid memory issues. There is no return value.

Usage

```
update_mode3_Tucker(m, d, params)
```

Arguments

m	A Tucker_model object created with mk_model
d	Input data object created with input_data
params	List of parameters created with get_model_params()

Examples

```
data.params <- get_data_params(c('decomp=Tucker'))
toy <- mk_toy(data.params)
train.data <- input_data$new(mode1.X=toy$mode1.X[,-1],
                             mode2.X=toy$mode2.X[,-1],
                             mode3.X=toy$mode3.X,
                             resp=toy$resp)
model.params <- get_model_params(c('decomp=Tucker'))
toy.model <- mk_model(train.data, model.params)
toy.model$rand_init(model.params)

update_mode3_Tucker(m=toy.model, d=train.data, params=model.params)
```

Index