Package 'MPTinR'

January 20, 2025

Type Package

Title Analyze Multinomial Processing Tree Models

Version 1.14.1

Description Provides a user-friendly way for the analysis of multinomial processing tree (MPT) models (e.g., Riefer, D. M., and Batchelder, W. H. [1988]. Multinomial modeling and the measurement of cognitive processes. Psychological Review, 95, 318-339) for single and multiple datasets. The main functions perform model fitting and model selection. Model selection can be done using AIC, BIC, or the Fisher Information Approximation (FIA) a measure based on the Minimum Description Length (MDL) framework. The model and restrictions can be specified in external files or within an R script in an intuitive syntax or using the context-free language for MPTs. The 'classical' .EQN file format for model files is also supported. Besides MPTs, this package can fit a wide variety of other cognitive models such as SDT models (see fit.model). It also supports multicore fitting and FIA calculation (using the snowfall package), can generate or bootstrap data for simulations, and plot predicted versus observed data.

License GPL (>= 2)

Depends R (>= 2.15.1)

Imports numDeriv, Brobdingnag, Rcpp, stats, utils

Suggests snowfall (>= 1.84), knitr

LinkingTo Rcpp, RcppEigen

LazyLoad yes

ByteCompile yes

VignetteBuilder knitr

NeedsCompilation yes

Author Henrik Singmann [aut, cre],
David Kellen [aut],
Quentin Gronau [aut],
Christian Mueller [ctb],
Akhil S Bhel [ctb]

Maintainer Henrik Singmann <singmann@gmail.com>

Repository CRAN

Date/Publication 2023-12-07 17:57:26

Contents

`MPTinR-package` *Analyze Multinomial Processing Tree Models*

Description

Provides a user-friendly way for the analysis of multinomial processing tree (MPT) models (e.g., Riefer, D. M., and Batchelder, W. H. [1988]. Multinomial modeling and the measurement of cognitive processes. Psychological Review, 95, 318-339) for single and multiple datasets. The main functions perform model fitting and model selection. Model selection can be done using AIC, BIC, or the Fisher Information Approximation (FIA) a measure based on the Minimum Description Length (MDL) framework. The model and restrictions can be specified in external files or within an R script in an intuitive syntax or using the context-free language for MPTs. The 'classical' .EQN file format for model files is also supported. Besides MPTs, this package can fit a wide variety of other cognitive models such as SDT models (see fit.model). It also supports multicore fitting and FIA calculation (using the snowfall package), can generate or bootstrap data for simulations, and plot predicted versus observed data.

Details

The DESCRIPTION file:

Package:	MPTinR
Type:	Package
Title:	Analyze Multinomial Processing Tree Models
Version:	1.14.1
Authors@R:	c(person("Henrik", "Singmann", role = c("aut", "cre"), email = "singmann@gmail.com"), person("Davi

Description:	Provides a user-friendly way for the analysis of multinomial processing tree (MPT) models (e.g., Riefer
License:	GPL (>= 2)
Depends:	R (>= 2.15.1)
Imports:	numDeriv, Brobdingnag, Rcpp, stats, utils
Suggests:	snowfall (>= 1.84), knitr
LinkingTo:	Rcpp, RcppEigen
LazyLoad:	yes
ByteCompile:	yes
VignetteBuilder:	knitr
NeedsCompilation:	yes
Packaged:	2021-07-13 10:44:49 UTC; singm
Author:	Henrik Singmann [aut, cre], David Kellen [aut], Quentin Gronau [aut], Christian Mueller [ctb], Akhil S
Maintainer:	Henrik Singmann <singmann@gmail.com>
Repository:	CRAN
Date/Publication:	2021-07-13 11:30:02 UTC

Index of help topics:

Further information is available in the following vignettes:

To fit MPT Models use `fit.mpt`, to fit other models use `fit.model` or `fit.mptinr` (which is called by the other two functions).

For model selection use `select.mpt`.

A helper function for writing model files is `check.mpt`

References

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339

Singmann, H., & Kellen, D. (2013). MPTinR: Analysis of multinomial processing tree models in R. *Behavior Research Methods*, 45(2), 560-575. doi:10.3758/s1342801202590

`bmpt.fia` *Compute FIA for MPTs*

Description

R-port of the function to compute FIA for MPT models by Wu, Myung, and Batchelder (2010a, 2010b). This function is essentially a copy of the original Matlab code to R (with significant parts moved to C++ and allowing for multicore functionality). Also, the order of input arguments is more R-like.

Usage

```
bmpt.fia(s, parameters, category, N, ineq0 = NULL, Sample = 2e+05,
        multicore = FALSE, split = NULL, mConst = NULL)
```

Arguments

`s`	see Details
`parameters`	see Details
`category`	see Details
`N`	see Details
`ineq0`	see Details
`Sample`	see Details
`multicore`	logical. Should fitting be distributed across several cores? Requires **snowfall** and initialized cluster. See also below.
`split`	`NULL` (the default) or integer specifying in how many separate calls to the C++ workhorse the integrant should be calculated. See below.
`mConst`	A constant which is added in the Monte Carlo integration to avoid numerical underflows and is later subtracted (after appropriate transformation). Should be a power of 2 to avoid unnecessary numerical imprecision.

Details

The following is the original description by Wu, Myung, & Batchelder (2010a) for their Matlab function. All changes to the original document are in squared brackets []:

This function computes the FIA complexity measure, C_FIA, using a Monte Carlo numerical integration algorithm. When inequality is present, sampling from the restricted parameter space is performed by rejection algorithm.

[...] [see References for References]

The following symbols are used in the body of the function:
S denotes number of parameters.
C denotes the number of categories.
M denotes the number of leaves in the tree.

The first input argument `s` is related to the string representation of the BMPT model. It can be obtained by replacing all categories in the string by the capital letter C and all branching probabilities by the lower case letter p.

The second input argument `parameters` is a row vector that assigns parameters or constants to the p's in the string `s`. Its length should be the same as the number of p's in `s`, and its elements correspond to the p's according to their order in `s`. Positive integer elements in `parameters` assign parameters to the corresponding p's, with the same integer denoting the same parameter. Constants are assigned to the p's using the negation of their values.

The [third] input argument `category` is a 1 by M vector assigning categories to the C's in the string 's' in the same way `parameters` assigns branching probabilities, except that only positive consecutive integers from 1 to `J`, the total number of categories, are allowed.

The [fourth] input argument `N` specifies the total sample size.

The [fifth] input argument `ineq0` assigns inequality constraints imposed on the parameters. It is a matrix with two columns. Each element denotes a parameter coded in the same way as in `parameters`. For each row, the parameter on the left column is constrained to be smaller than that on the right column. The number of rows is determined by the total number of simple inequality constraints of the form `theta_1 < theta_2` in the model. [Default is `NULL` corresponding to no inequality restrictions.]

The last input argument 'Sample' specifies the number of random samples to be drawn in the Monte Carlo algorithm. [Default is 200000.]

[For returned values see Value]

It should be noted that 'lnconst' can be computed analytically free of Monte Carlo error on a case by case basis described below. For this reason, the users can calculate `C_FIA` [see Wu, Myung & Batchelder, 2010a; Equation 7] by adding `(S/2)*ln*(N/(2*pi))`, `lnInt` and their hand-calculated `lnconst` to minimize the Monte Carlo errors. [In our experience this error is rather low and negligible.]

A sequence of inequalities `theta_1 < theta_2 < ... < theta_k` reduces the parameter space to its `1/k!`, so in this case `lnconst` should be `-ln * (k!)`. In general, any combination of inequality constraints specifies a union of subsets of the parameter space, each satisfying some sequence of inequalities. For example, the subspace defined by `theta_1 < theta_2` and `theta_3 < theta_2` is a union of two subspaces, one satisfying `theta_1 < theta_3 < theta_2` and the other `theta_3 < theta_1 < theta_2`, so the proportion is given by `2 * (1/3!) = 1/3`.

A coding example:
Suppose that for model 1HTM-5c of source monitoring [see Wu et al., 2010a] , the sample sizes of source A, source B and new items are 300, 300 and 400, respectively and the inequality constraint of d_1 < d_2 is imposed. In this case, the six input arguments should be specified as follows:
s = 'ppppCpCCppCCCppCpCCppCCCppCCC';
parameters = c(-.6,-.5,1,2,5,4,5,1,3,5,4,5,4,5); [adapted for R]
ineq0 = matrix(c(2,3), 1,2); [adapted for R]
category = c(1,1,2,1,2,3,5,4,5,4,5,6,7,8,9); [adapted for R]
N = 1000;

Another coding example:
For the pair-clustering model in Batchelder and Riefer (1999, Figure 1), suppose in a pair-clustering experiment there are 300 pairs of words and 100 singletons, the six input arguments should be specified as follows:
s = 'pppCCppCCpCCpCC'; parameters = c(-.75,1,2,3,3,3,3); [adapted for R]
ineq0 = NULL; [adapted for R]
category = c(1,4,2,3,3,4,5,6); [adapted for R]
N = 400;

[For more examples, see Examples]

Since MPTinR version 1.1.3 the Monte Carlo integration is performed in C++ using **RcppEigen**. With the default arguments, one instance of the C++ workhorse is called. To call multiple instances of the C++ workhorse, you can use the `split` argument (which can be useful to replicate results obtained with `multicore = TRUE` as described below). Note, that each time before calling the C++ code, the seed is set (the set of random seeds are generated before calling the function for the first time).

Multicore functionality is achieved via **snowfall** which needs to be loaded and a cluster initialized via `sfInit` when setting `multicore = TRUE`. When `split = NULL` (the default), the `Samples` will be evenly distributed on the different cores (using `sfClusterSplit`), so that only one instance of the underlying C++ workhorse is called on each core. Setting `split` to non-`NULL` will produce as many instances (distributed across cores). Note that in order to obtain comparable results (as snowfall uses load balancing), the random seed is set (at each core) before calling each instance of the C++ workhorse. This allows to replicate results obtained via multicore in a non-multicore environment when seting `split` appropriately (and `set.seed` beforehands).

Value

[A named vector:]

The first output argument `CFIA` gives the FIA complexity value of the model.

The second [and third] output argument `CI` gives the Monte Carlo confidence interval of `CFIA`. [`CI.l`, gives the lower, `CI.u`, the upper bound of the interval].

The [fourth] output argument `lnInt` gives the log integral term in `C_FIA` [see Wu, Myung & Batchelder, 2010a; Equation 7] for models without inequality constraints. When inequality constraints are present, `lnInt` does not take into account the change in the normalizing constant in the proposal distribution and must be adjusted with the output argument `lnconst`.

The [fifth and sixth] output argument [`CI.lnint`] gives the Monte Carlo confidence interval of `lnInt`. [.l = lower & .u = upper bound of the CI]

When inequality constraints are present, the [seventh] output argument `lnconst` serves as an adjustment of 'lnInt'. It estimates the logarithm of the proportion of parameter space [0,1]^S that satisfies those inequality constraints, and the log integral term is given by lnInt+lnconst.

The next [two] output argument [`CI.lnconst`] give the Monte Carlo confidence interval of 'lnconst'. [.l = lower & .u = upper bound of the CI]

Note

The R version of the code should now (after moving the code to RcppEigen) be considerably faster than the Matlab version of this code.

Author(s)

The original Matlab code was written by Hao Wu, Jay I. Myung, and William H. Batchelder.
This code was ported to R by Henrik Singmann and David Kellen. RcppEigen was added by Henrik Singmann and Christian Mueller. Multicore functionality was added by Henrik Singmann.

References

Wu, H., Myung, J.I., & Batchelder, W.H. (2010a). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010b). On the minimum description length complexity of multinomial processing trees. *Journal of Mathematical Psychology*, 54, 291-303.

See Also

`fit.mpt` for the main function of MPTinR.
`get.mpt.fia` for a convenient wrapper of this function.

Examples

```
## Not run:
# The following example is the code for the first example in Wu, Myung & Batchelder (2010a, pp. 280)
# The result should be something like: CFIA = 12.61... or 12.62..., CI = 12.61... - 12.62....
# Executing this command can take a while.

bmpt.fia(s = "ppppCpCCppCCCppCpCCppCCCppCCC",
parameters = c(-0.5, -0.5, 3, 2, 5, 1, 5, 4, 2, 5, 1, 5, 1, 5),
category = c(1,1,2,1,2,3,5,4,5,4,5,6,7,8,9),
  N = 1000, ineq0 = matrix(c(4,3),1,2))

bmpt.fia(s = "ppppCpCCppCCCppCpCCppCCCppCCC",
parameters = c(-0.5, -0.5, 3, 2, 5, 1, 5, 4, 2, 5, 1, 5, 1, 5),
category = c(1,1,2,1,2,3,5,4,5,4,5,6,7,8,9),
  N = 1000, ineq0 = matrix(c(4,3),1,2), mConst = 2L^8)

## End(Not run)
```

check.mpt *Check construction of MPT models.*

Description

A helper function which can aid in the process of constructing a MPT model file for MPTinR. It will check if the probabilities in each trees sum to 1 (if so, a tree is well constructed). If probabilities do not sum to 1, check.mpt will return for which trees. Furthermore, it will return the number of parameters and their names (helpful in spotting typos), the number of categories and the number of dfs the model provides. Finally, you can also pass restrictions as an argument and will receive the number and names of the parameters after restrictions are applied.

Usage

```
check.mpt(model.filename, restrictions.filename = NULL, model.type = c("easy", "eqn"))
```

Arguments

model.filename	A character vector specifying the location and name of the model file.
restrictions.filename	NULL or a character vector specifying the location and name of the restrictions file. Default is NULL which corresponds to no restrictions.
model.type	Character vector specifying whether the model file is formatted in the *easy* format ("easy"; i.e., each line represents all branches corresponding to a response category) or the traditional EQN syntax ("eqn"; see e.g., Stahl & Klauer, 2007). If the model filename ends with ".eqn" or ".EQN" the model is automatically treated as an EQN file.

Details

As default, check.mpt expects a model file in the easy format, but if the filename ends with .eqn or .EQN check.mpt will expect the EQN format.

In case of inequality restrictions, the original parameters which are inequality restricted are replaced with dummy parameters starting with hankX. When using fit.mpt you will not notice this, as the output only shows the original parameters. In contrast, check.mpt removes the original parameters and shows the dummy parameters called hankX. Note that this does not change the number of parameters in the model.

For EQN model files, the output also contains two slots giving the ordering of trees (eqn.order.trees) as well as categories (eqn.order.categories).

Value

A list with

`probabilites.eq.1`
: A logical value indicating whether or not the probabilities in each tree sum to 1. If `FALSE`, a warning is shown indicating in which trees the probabilities do not sum to 1.

`n.trees`
: Number of trees in the model.

`n.model.categories`
: Total number of categories expected in a dataset for that model.

`n.independent.categories`
: Number of independent response categories (i.e., independent data points) the model provides (i.e., `n.model.categories - n.trees`). The number of parameters can not be higher than this value for a model to be identifiable.

`n.params`
: Number of parameters in the model.

`parameters`
: Names of parameters in the model.

`eqn.order.trees`
: Order of trees in EQN model files (omitted if model is not an EQN file).

`eqn.order.categories`
: Order of trees in EQN model files (omitted if model is not an EQN file).

If restrictions are present, the `n.params` and `parameters` are displayed for the unrestricted model (`orig.model`) as well as for the restricted model (`restr.model`).

See Also

`fit.mpt`

Examples

```
# model of example 1 from example(fit.mpt)
model1 <- system.file("extdata", "rb.fig1.model", package = "MPTinR")
check.mpt(model1)

#model 1 in eqn format
model1.eqn <- system.file("extdata", "rb.fig1.model.eqn", package = "MPTinR")
check.mpt(model1.eqn)



#models of example 2 from example(fit.mpt)
model2 <- system.file("extdata", "rb.fig2.model", package = "MPTinR")
check.mpt(model2)

model2r.r.eq <- system.file("extdata", "rb.fig2.r.equal", package = "MPTinR")
check.mpt(model2, model2r.r.eq)

model2r.c.eq <- system.file("extdata", "rb.fig2.c.equal", package = "MPTinR")
check.mpt(model2, model2r.c.eq)
```

d.broeder *Broeder & Schuetz (2009) Experiment 3*

Description

The data from Broeder & Schuetz (2009) Experiment 3, used as an example in MPTinR

Usage

```
data(d.broeder)
```

References

Broeder, A., & Schuetz, J. (2009). Recognition ROCs are curvilinear-or are they? On premature arguments against the two-high-threshold model of recognition. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 35(3), 587. doi:10.1037/a0015279

fit.model *Fit cognitive models for categorical data using model files*

Description

`fit.model` fits MPT and other cognitive models for categorical data (e.g., SDT models) that can be specified in a model file.

Usage

```
fit.model(
data,
model.filename,
restrictions.filename = NULL,
n.optim = 5,
fia = NULL,
ci = 95,
starting.values = NULL,
lower.bound = 0,
upper.bound = 1,
output = c("standard", "fia", "full"),
reparam.ineq = TRUE,
fit.aggregated = TRUE,
sort.param = TRUE,
show.messages = TRUE,
model.type = c("easy", "eqn", "eqn2"),
multicore = c("none", "individual", "n.optim", "fia"), sfInit = FALSE, nCPU = 2,
control = list(),
```

```
    use.gradient = TRUE, use.hessian = FALSE, check.model = TRUE,
        args.fia = list(), numDeriv = TRUE
    )
```

Arguments

data	Either a *numeric* vector for individual fit or a *numeric* matrix or data.frame for multi-individual fit. The data on each position (column for multi-individual fit) must correspond to the respective line in the model file. Fitting for multiple individuals can be parallelized via multicore.
model.filename	A character vector specifying the location and name of the model file.
restrictions.filename	NULL or a character vector or a list of characters. The default is NULL which corresponds to no restrictions. A character vector specifies the location or name of the restrictions file. A list of characters contains the restrictions directly. See Details and Examples.
n.optim	Number of optimization runs. Can be parallelized via multicore. Default is 5. If the number is high, fitting can take long time for *large* models.
fia	Number of random samples to be drawn in the Monte Carlo algorithm to estimate the Fisher Information Approximation (FIA) for MPTs only. See Details at fit.mpt
ci	A scalar corresponding to the size of the confidence intervals for the parameter estimates. Default is 95 which corresponds to 95% confidence intervals.
starting.values	A vector, a list, or NULL (the default). If NULL starting values for parameters are randomly drawn from a uniform distribution with the interval (0.1 - 0.9). See Details for the other options.
output	If "full" fit.mpt will additionally return the return values of nlminb and the Hessian matrices. (If "fia", fit.mpt will additionally return the results from get.mpt.fia (if fia not equal NULL).)
reparam.ineq	Logical. Indicates whether or not inequality restrictions (when present in the model file) should be enforced while fitting the model. If TRUE (the default) inequality restricted parameters will be reparameterized, if FALSE not. Probably irrelevant for none MPTs.
fit.aggregated	Logical. Only relevant for multiple datasets (i.e., matrix or data.frame). Should the aggregated dataset (i.e., data summed over rows) be fitted? Default (TRUE) fits the aggregated data.
sort.param	Logical. If TRUE, parameters are alphabetically sorted in the parameter table. If FALSE, the first parameters in the parameter table are the non-restricted ones, followed by the restricted parameters. Default is TRUE.
show.messages	Logical. If TRUE the time the fitting algorithms takes is printed to the console.
model.type	Character vector specifying whether the model file is formatted in the easy way ("easy"; i.e., each line represents all branches corresponding to a response category) or the traditional EQN syntax ("eqn" or "eqn2"; see Details and e.g., Stahl & Klauer, 2007). If model.filename ends with .eqn or .EQN, model.type is automatically set to "eqn". Default is "easy".

multicore	Character vector. If not "none", uses snowfall for parallelization (which needs to be installed separately via install.packages(snowfall)). If "individual", parallelizes the optimization for each individual (i.e., data needs to be a matrix or data.frame). If "n.optim", parallelizes the n.optim optimization runs. If not "none" (e.g., "fia") calculation of FIA is parallelized (if FIA is requested). Default is "none" which corresponds to no parallelization. Note that you need to initialize snowfall in default settings. See sfInit and Details.
sfInit	Logical. Relevant if multicore is not "none". If TRUE, fit.mpt will initialize and close the multicore support. If FALSE, (the default) assumes that sfInit() was initialized before. See Details.
nCPU	Scalar. Only relevant if multicore is not "none" and sfInit is TRUE. Number of CPUs used by snowfall. Default is 2.
lower.bound	numeric scalar or vector. Can be used in fit.model to set the lower bounds of the parameter space. See Details.
upper.bound	numeric scalar or vector. Can be used in fit.model to set the upper bounds of the parameter space. See Details.
control	list containing control arguments passed on to nlminb. See there.
use.gradient	logical. Whether or not the symbolically derived function returning the gradient should be used for fitting. Default is TRUE meaning gradient function is used.
use.hessian	logical. Whether or not the symbolically derived function returning the Hessian matrix should be used for fitting. Default is FALSE meaning hessian function is not used.
check.model	logical. Should model be checked with random values whether or not the expected values sum to one per tree? Default is TRUE. (This also controls whether other model checks during optimization are performed. If FALSE the most permissive fitting is performed.)
args.fia	named list of further arguments passed to get.mpt.fia, such as mConst to avoid numerical problems in the FIA function.
numDeriv	logical. Should the Hessian matrix of the maximum likelihood estimates be estimated numerically using numDeriv::hessian in case it cannot be estimated analytically? This can be extremely time and memory consuming for larger models. Default is TRUE.

Details

This functions should be used when fitting a model that is not an MPT model or when fitting using fit.mpt fails. For fitting MPT models and information on fitting MPT models see fit.mpt.

The model file for non-MPT models should be of the easy format. That is the ordinal number or rank of each line should correspond to this column/position in the data object. Model files can contain any visible function (i.e., including self-defined functions). However, note that the derivation that is needed for the gradient and Hessian function can only be done for those functions that D can handle. If derivation fails a warning will be given and fitting will be done without gradient and/or Hessian function.

Equations that correspond to one item type/category must be not be separated by an empty line. Equations that do not correspond to the same item type/category must be separated by at least one empty line.

Note that names of parameters in the model file should NOT start with hank. Variables with these names can lead to unforeseen problems as variables starting with these letters are internally used.

The restrictions file may contain (sequential) equality (i.e., =) and inequality (i.e., <) restrictions (see fit.mpt for more general info on the restrictions files). Note that inequality restrictions usually will lead to catastrophic results when used for non-MPT models. Our recommendation: Do never use inequality restrictions for non-MPT models. Equality restrictions or fixing parameters should be no problem though.

For equality restrictions, the equality restricted parameters are simply exchanged with their restrictions (i.e., another parameter or a number) before the fitting.

Restrictions or model files can contain comments (i.e., everything to the right of a # will be ignored; new behavior since version 0.9.2)

Both models and restrictions can be specified as textConnections instead of as external files (see examples). Note that textConnections get "consumed" so you may need to specify them each time you fit a model using a connection (see Examples for how to avoid this).

Confidence intervals (CI) are based on the Hessian matrix produced by the symbolically derived function for the Hessian (i.e., the second derivative of the likelihood function). If it is based on a numerically estimated Hessian, a warning will be given.

To set the starting values for the fitting process (e.g., to avoid local minima) one can set starting.values to a vector of length 2 and n.optim > 1. Then, starting values are randomly drawn from a uniform distribution from starting.values[1] to starting.values[2].

Alternatively, one can supply a list with two elements to starting.values. Both elements need to be either of length 1 or of length equal to the number of parameters (if both are of length 1, it is the same as if you supply a vector of length 2). For each parameter n (in alphabetical order), a starting value is randomly drawn from a uniform distribution starting.values[[1]][n] to starting.values[[2]][n] (if length is 1, this is the border for all parameters).

The least interesting option is to specify the starting values individually by supplying a vector with the same length as the number of parameters. Starting values must be ordered according to the alphabetical order of the parameter names. Use check.mpt for a function that returns the alphabetical order of the parameters. If one specifies the starting values like that, n.optim will be set to 1 as all other values would not make any sense (the optimization routine will produce identical results with identical starting values).

The lower.bound and upper.bound needs to be of length 1 or equal to the number of free parameters. If length > 1, parameters are mapped to the bounds in alphabetic order of the parameters. Use check.mpt to obtain the alphabetical order of parameters for your model.

This function is basically a comfortable wrapper for fit.mptinr producing the appropriate objective, gradient, hessian, and prediction function from the model equations (passed via model.filename) whilst allowing for custom lower or upper bounds on the parameters. You can specify whether or not gradient or hessian function should be used for fitting with use.gradient or use.hessian, respectively.

Multicore fitting is achieved via the snowfall package and needs to be initialized via sfInit. As initialization needs some time, you can either initialize multicore facilities yourself using sfInit() and setting the sfInit argument to FALSE (the default) or let MPTinR initialize multicore facilities by setting the sfInit argument to TRUE. The former is recommended as initializing snowfall takes some time and only needs to be done once if you run fit.mpt multiple times. If there are

any problems with multicore fitting, first try to initialize snowfall outside MPTinR (e.g., sfInit(parallel=TRUE, cpus=2)). If this does not work, the problem is not related to MPTinR but to snowfall (for support and references visit: https://www.imbi.uni-freiburg.de/parallel/). Note that you should *close* snowfall via sfStop() after using MPTinR.

Value

For individual fits (i.e., data is a vector) a list containing one or more of the following components from the best fitting model:

goodness.of.fit
: A data.frame containing the goodness of fit values for the model. Log.Likelihood is the Log-Likelihood value. G.Squared, df, and p.value are the G^2 goodness of fit statistic.

information.criteria
: A data.frame containing model information criteria based on the G^2 value. The FIA values(s) are presented if fia is not NULL.

model.info
: A data.frame containing other information about the model. If the rank of the Fisher matrix (rank.fisher) *does not* correspond to the number of parameters in the model (n.parameters) this indicates a serious issue with the identifiability of the model. A common reason is that one of the parameter estimates lies on the bound of the parameter space (i.e., 0 or 1).

parameters
: A data.frame containing the parameter estimates and corresponding confidence intervals. If a restriction file was present, the restricted parameters are marked.

data
: A list of two matrices; the first one (observed) contains the entered data, the second one (predicted) contains the predicted values.

For multi-dataset fits (i.e., data is a matrix or data.frame) a list with similar elements, but the following differences:
The first elements, goodness.of.fit, information.criteria, and model.info, contain the same information as for individual fits, but each are lists with three elements containing the respective values for: each individual in the list element individual, the sum of the individual values in the list element sum, and the values corresponding to the fit for the aggregated data in the list element aggregated.
parameters is a list containing:

individual
: A 3-dimensional array containing the parameter estimates ([,1,]), confidence intervals [,2:3,], and, if restrictions not NULL, column 4 [,4,] is 0 for non-restricted parameters, 1 for equality restricted parameters, and 2 for inequality restricted parameters. The first dimension refers to the parameters, the second to the information on each parameter, and the third to the individual/dataset.

mean
: A data.frame with the mean parameter estimates from the individual estimates. No confidence intervals can be provided for these values.

aggregated
: A data.frame containing the parameter estimates and corresponding confidence intervals for the aggregated data. If a restriction file was present, the restricted parameters are marked.

The element data contains two matrices, one with the observed, and one with the predicted data (or is a list containing lists with individual and aggregated observed and predicted data).

If n.optim > 1, the summary of the vector (matrix for multi-individual fit) containing the Log-Likelihood values returned by each run of optim is added to the output: fitting.runs

When output == "full" the list contains the additional items:

optim.runs	A list (or list of lists for multiple datasets) containing the outputs from all runs by nlminb (including those runs produced when fitting did not converge)
best.fits	A list (or list of lists for multiple datasets) containing the outputs from the runs by nlminb that had the lowest likelihood (i.e., the successful runs)
hessian	A list containing the Hessian matrix or matrices of the final parameter estimates.

Note

Warnings may relate to the optimization routine (e.g., Optimization routine [...] did not converge successfully).
In these cases it is recommended to rerun the model fitting to check if the results are stable.

The likelihood returned does not include the factorial constants of the multinomial probability-mass functions.

All (model or restriction) files should end with an empty line, otherwise a warning will be shown.

Author(s)

Henrik Singmann and David Kellen.

References

Broeder, A., & Schuetz, J. (2009). Recognition ROCs are curvilinear-or are they? On premature arguments against the two-high-threshold model of recognition. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 35(3), 587. doi:10.1037/a0015279

Wickens, T. D. (2002). *Elementary Signal Detection Theory*. Oxford; New York: Oxford University Press.

See Also

check.mpt for a function that can help in constructing models.

fit.mptinr for a function that can fit arbitrary objective functions.

fit.mpt for the function to fit MPTs (it should be slightly faster for MPTs).

roc6 for more examples fitting different SDT models.

Examples

```
## Not run:

####################################
## Fit response-bias or payoff ROC ##
####################################

# Example from Broder & Schutz (2009)
```

```
# We fit the data from the 40 individuals from their Experiment 3
# We fit three different models:
# 1. Their SDT Model: br.sdt
# 2. Their 2HTM model: br.2htm
# 3. A restricted 2HTM model with Dn = Do: br.2htm.res
# 4. A 1HTM model (i.e., Dn = 0): br.1htm

data(d.broeder, package = "MPTinR")
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")


# We specify the SDT model in the code using a textConnection.
# However, textConnection is only called in the function call on the string.

m.sdt <- "
1-pnorm((cr1-mu)/ss)
pnorm((cr1-mu)/ss)

1-pnorm(cr1)
pnorm(cr1)

1-pnorm((cr2-mu)/ss)
pnorm((cr2-mu)/ss)

1-pnorm(cr2)
pnorm(cr2)

1-pnorm((cr3-mu)/ss)
pnorm((cr3-mu)/ss)

1-pnorm(cr3)
pnorm(cr3)

1-pnorm((cr4-mu)/ss)
pnorm((cr4-mu)/ss)

1-pnorm(cr4)
pnorm(cr4)

1-pnorm((cr5-mu)/ss)
pnorm((cr5-mu)/ss)

1-pnorm(cr5)
pnorm(cr5)
"

# How does the model look like?
check.mpt(textConnection(m.sdt))

# fit the SDT (unequal variance version)
br.uvsdt <- fit.model(d.broeder, textConnection(m.sdt),
   lower.bound = c(rep(-Inf, 5), 0, 1), upper.bound = Inf)
```

```
# Is there any effect of studying the items?
br.uvsdt.2 <- fit.model(d.broeder, textConnection(m.sdt),
restrictions.filename = list("mu = 0", "ss = 1"),
lower.bound = -Inf, upper.bound = Inf)

(diff.g2 <- br.uvsdt.2[["goodness.of.fit"]][["sum"]][["G.Squared"]] -
br.uvsdt[["goodness.of.fit"]][["sum"]][["G.Squared"]])
(diff.df <- br.uvsdt.2[["goodness.of.fit"]][["sum"]][["df"]] -
br.uvsdt[["goodness.of.fit"]][["sum"]][["df"]])
1 - pchisq(diff.g2, diff.df)

# fit the equal variance SDT model:
br.evsdt <- fit.model(d.broeder, textConnection(m.sdt),
lower.bound = c(rep(-Inf, 5), 0), upper.bound = Inf,
restrictions.filename = list("ss = 1"))

# fit the MPTs (see also ?fit.mpt).
# In contrast to ?fit.mpt we specify the restrictions using a textConnection or a list!
br.2htm <- fit.mpt(d.broeder, m.2htm)
br.2htm.res <- fit.mpt(d.broeder, m.2htm, textConnection("Do = Dn"))
br.1htm <- fit.mpt(d.broeder, m.2htm, list("Dn = 0"))

select.mpt(list(uvsdt = br.uvsdt, evsdt = br.evsdt, two.htm = br.2htm,
two.htm.res = br.2htm.res, one.htm = br.1htm), output = "full")

# the restricted 2HTM "wins" for individual data (although evsdt does not perform too bad),
# but the 2htm and restricted 2htm restricted "win" for aggregated data.


###################################
## Fit confidence rating ROC SDT ##
###################################
#(see ?roc6 for more examples)

# We fit example data from Wickens (2002, Chapter 5)
# The example data is from Table 5.1, p. 84
# (data is entered in somewhat different order).

# Note that criteria are defined as increments to
# the first (i.e., leftmost) criterion!
# This is the only way to do it in MPTinR.

# Data
dat <- c(47, 65, 66, 92, 136, 294, 166, 161, 138, 128, 63, 43)

# UVSDT
m.uvsdt <- "
pnorm(cr1, mu, sigma)
pnorm(cr1+cr2, mu, sigma) - pnorm(cr1, mu, sigma)
pnorm(cr3+cr2+cr1, mu, sigma) - pnorm(cr2+cr1, mu, sigma)
pnorm(cr4+cr3+cr2+cr1, mu, sigma) - pnorm(cr3+cr2+cr1, mu, sigma)
pnorm(cr5+cr4+cr3+cr2+cr1, mu, sigma) - pnorm(cr4+cr3+cr2+cr1, mu, sigma)
1 - pnorm(cr5+cr4+cr3+cr2+cr1, mu, sigma)
```

```
pnorm(cr1)
pnorm(cr2+cr1) - pnorm(cr1)
pnorm(cr3+cr2+cr1) - pnorm(cr2+cr1)
pnorm(cr4+cr3+cr2+cr1) - pnorm(cr3+cr2+cr1)
pnorm(cr5+cr4+cr3+cr2+cr1) - pnorm(cr4+cr3+cr2+cr1)
1 - pnorm(cr5+cr4+cr3+cr2+cr1)
"
check.mpt(textConnection(m.uvsdt))

# Model fitting
(cr_sdt <- fit.model(dat, textConnection(m.uvsdt),
            lower.bound=c(-Inf, rep(0, 5), 0.1), upper.bound=Inf))

# To obtain the criteria (which match those in Wickens (2002, p. 90)
# obtain the cumulative sum:

cumsum(cr_sdt$parameters[paste0("cr",1:5), 1, drop = FALSE])


## End(Not run)
```

`fit.mpt` *Function to fit MPT models*

Description

`fit.mpt` fits *binary* multinomial processing tree models (MPT models; e.g., Riefer & Batchelder, 1988) from an external model file and (optional) external restrictions using the general-purpose quasi-Newton box-constraint optimization routine provided by Byrd et al. (1995). Additionally, measures for model selection (AIC, BIC, FIA) can be computed.

Usage

```
fit.mpt(
data,
model.filename,
restrictions.filename = NULL,
n.optim = 5,
fia = NULL,
ci = 95,
starting.values = NULL,
output = c("standard", "fia", "full"),
reparam.ineq = TRUE,
fit.aggregated = TRUE,
sort.param = TRUE,
show.messages = TRUE,
model.type = c("easy", "eqn", "eqn2"),
```

```
multicore = c("none", "individual", "n.optim", "fia"), sfInit = FALSE, nCPU = 2,
control = list(), args.fia = list(), numDeriv = TRUE
)
```

Arguments

data	Either a *numeric* vector for individual fit or a *numeric* matrix or data.frame for multi-individual fit. The data on each position (column for multi-individual fit) must correspond to the respective line in the model file. For EQN model files, the required order is given by an alphabetic ordering of first model names than categories. This order in terms of the names in the EQN file can be obtained through check.mpt. Fitting for multiple individuals can be parallelized via multicore.
model.filename	A character vector specifying the location and name of the model file.
restrictions.filename	NULL or a character vector or a list of characters. The default is NULL which corresponds to no restrictions. A character vector specifies the location or name of the restrictions file. A list of characters contains the restrictions directly. See Details and Examples.
n.optim	Number of optimization runs. Can be parallelized via multicore. Default is 5. If the number is high, fitting can take long for *large* models.
fia	Number of random samples to be drawn in the Monte Carlo algorithm to estimate the Fisher Information Approximation (FIA), a minimum description length based measure of model complexity (see Wu, Myung & Batchelder, 2010). The default is NULL which corresponds to no computation of the FIA. Reasonable values (e.g., > 200000) can lead to long computation times (minutes to hours) depending on the size of the model. See Details.
ci	A scalar corresponding to the size of the confidence intervals for the parameter estimates. Default is 95 which corresponds to 95% confidence intervals.
starting.values	A vector, a list, or NULL (the default). If NULL starting values for parameters are randomly drawn from a uniform distribution with the interval (0.1 - 0.9). See Details of fit.mptinr for the other options.
output	If "fia", fit.mpt will additionally return the results from get.mpt.fia (if fia not equal NULL). If "full" fit.mpt will additionally return the results from get.mpt.fia and the output of nlminb and the Hessian matrix/matrices.
reparam.ineq	Logical. Indicates whether or not inequality restrictions (when present in the model file) should be enforced while fitting the model. If TRUE (the default) inequality restricted parameters will be reparameterized, if FALSE not. See Details.
fit.aggregated	Logical. Only relevant for multiple datasets (i.e., matrix or data.frame). Should the aggregated dataset (i.e., data summed over rows) be fitted? Default (TRUE) fits the aggregated data.
sort.param	Logical. If TRUE, parameters are alphabetically sorted in the parameter table. If FALSE, the first parameters in the parameter table are the non-restricted ones, followed by the restricted parameters. Default is TRUE.

show.messages	Logical. If TRUE the time the fitting algorithms takes is printed to the console.
model.type	Character vector specifying whether the model file is formatted in the easy way ("easy"; i.e., each line represents all branches corresponding to a response category) or the traditional EQN syntax ("eqn" or "eqn2"; see Details and e.g., Stahl & Klauer, 2007). If model.filename ends with .eqn or .EQN, model.type is automatically set to "eqn". Default is "easy".
multicore	Character vector. If not "none", uses snowfall for parallelization (which needs to be installed separately via install.packages(snowfall)). If "individual", parallelizes the optimization for each individual (i.e., data needs to be a matrix or data.frame). If "n.optim", parallelizes the n.optim optimization runs. If not "none" (e.g., "fia") calculation of FIA is parallelized (if FIA is requested). Default is "none" which corresponds to no parallelization. Note that you need to initialize snowfall in default settings. See sfInit and Details.
sfInit	Logical. Relevant if multicore is not "none". If TRUE, fit.mpt will initialize and close the multicore support. If FALSE, (the default) assumes that sfInit() was initialized before. See Details.
nCPU	Scalar. Only relevant if multicore is not "none" and sfInit is TRUE. Number of CPUs used by snowfall. Default is 2.
control	list containing control arguments passed on to nlminb. See there.
args.fia	named list of further arguments passed to get.mpt.fia, such as mConst to avoid numerical problems in the FIA function.
numDeriv	logical. Should the Hessian matrix of the maximum likelihood estimates be estimated numerically using numDeriv::hessian in case it cannot be estimated analytically? This can be extremely time and memory consuming for larger models. Default is TRUE.

Details

The model file is either of the easy format or the "classical" EQN format (see below).
In the easy format (the default) the model file contains all trees of the model. Trees are separated by at least one empty line. Everything to the right of a hash (#) is ignored (this behavior is new since version 0.9.2). Lines starting with a # are treated as empty. Each line in each tree corresponds to all branches of this tree (concatenated by a +) that correspond to one of the possible response categories. The position of each line must correspond to the position of this response category in the data object (for multi-individual fit to the respective column).

For EQN model files the order of the data does NOT correspond to the order in the model file, but to the order given by first sorting the tree names alphabetically/numerically and than the category names within the trees. As this is often difficult to see if the EQN files contains names and not numbers for trees and categories, check.mpt returns the order of both trees and categories for EQN model files.

The difference between both types of EQN format ("eqn" or"eqn2") is the way the first line of the model file is treated. If model.file is set to "eqn", MPTinR will ignore the first line of the model file and will read the rest of the file (as does multiTree; Moshagen, 2010). If model.file is set to "eqn2" MPTinR will only read as many lines as indicated in the first line of the EQN model file (as does e.g., HMMTree; Stahl & Klauer, 2007). As default fit.mpt expects the easy format, but if the filename ends with .eqn or .EQN and model.type is "easy", model.type is set to "eqn"

For the EQN format consult one of the corresponding papers (see e.g., Moshagen, 2010; Stahl & Klauer, 2007). The positions in the data object (number of column for multi-individual fit) must correspond to the category number in the EQN file.

Note that names of parameters in the model file should not start with `hank.`. Variables with these names can lead to unforeseen problems as variables starting with these letters are internally used. Furthermore, any `reserved` names (e.g., `NA`) are not allowed in model files of any types (i.e., also not as category labels in .eqn files). All names in models need to be valid R variable names (see `make.names`).

The restrictions file may contain (sequential) equality (i.e., =) and inequality (i.e., <) restrictions and must adhere to the following rules:
1. Inequalities first.
2. If a variable appears in an inequality restriction, it can not be on the left hand side (LHS) of any further restriction.
3. If a variable appears on the right hand side (RHS) of an equality restriction, it can not appear on LHS of an equality restriction.
Note that only "<" is supported as inequality operator but not ">"!
Examples of restrictions are (the following could all appear in one restrictions file):

```
D1 < D2 < D3
D4 = D3
B1 = B3 = 0.3333
X4 = X5 = D3
```

Restrictions file may contain comments (i.e., everything to the right of a # will be ignored; new behavior since version 0.9.2)

Restrictions can also be specified in line as a list. The same restrictions as the one above as a `list` would be `list("D1 < D2 < D3", "D4 = D3", "B1 = B3 = 0.3333", "X4 = X5 = D3")` (simply use this `list` as the `restrictions.filename` argument).

For equality restrictions, the equality restricted parameters are simply exchanged with their restrictions before the fitting.
For inequality restricted parameters, the model is reparameterized so that only the rightmost parameter of an inequality restriction remains the original parameter. Each instance of the other parameters in this restriction is replaced by the product of the rightmost parameter and dummy parameters (see Knapp & Batchelder, 2004). This procedure (which is equivalent to method A described in Knapp & Batchelder, 2004) leads to an equivalent model (although the binary MPT structure is not apparent in the resulting equations).
To prohibit this reparameterization (i.e., if the inequality restrictions hold without reparameterization), you can set `reparam.ineq` to `FALSE`. This can be useful for obtaining the FIA (see examples in Wu, Myung, & Batchelder, 2010).

Both models and restrictions can be specified as `textConnection`s instead of as external files.
Furthermore, restrictions can be specified directly as a `list` containing the restrictions (quoted, i.e. as characters).
`fit.model` contains additional examples showing model and restrictions specification within the code.

Note that when setting some parameters equal and also restricting their order, the parameters set equal which are not the rightmost element in the order (i.e., inequality) restriction, are computed correctly, but are marked as inequality restricted instead of equality restricted in the output (this did not work at all before v1.0.1). An example: For the restrictions `list("G2 < G3 < G5", "G1 = G2",`

"G4 = G5"), G1 would be computed correctly, but marked as inequality restricted. In contrast, G4 would be marked as equal to G5 (and also computed correctly).

To obtain a measure of the model's complexity beyond the number of parameters (and taking inequality restrictions into account), set fia to a (reasonably high) scalar integer (i.e., a number). Then, fit.mpt will obtain the Fisher Information Approximation (FIA), a Minimum Description Length (MDL) based measure of model complexity, using the algorithm provided by Wu, Myung, & Batchelder (2010a, 2010b) ported from Matlab to R. When performing model-selection, this measure is superior to other methods such as the Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) which basically only take the number of parameters into account.
To get the FIA, fit.mpt performs the following steps:
1. The representation of the model as equations is transformed into the string representation of the model in the context-free language of MPT models (L-BMPT; Purdy & Batchelder, 2009). For this step to be successful it is *absolutely necessary* that the equations representing the model perfectly map the tree structure of the MPT. That is, the model file is only allowed to contain parameters, their inverse (e.g., Dn and (1 - Dn)) and the operators + and *, but nothing else. Simplifications of the equations will seriously distort this step. Similarly, unnecessary brackets will distort the results. Brackets must only be used to indicate the inverse of a parameter (i.e. (1 - parameter)). This step is achieved by make.mpt.cf.
2. The context free representation of the model is then fed into the MCMC function computing the FIA (the port of BMPTFIA provided by Wu, Myung & Batchelder (2010a), see bmpt.fia).
(Actually, both steps are achieved by a call to get.mpt.fia)

Note that FIA can sometimes be non-consistent (i.e., larger FIA penalty values for restricted versions of a model than for the superordinate model; see Navarro, 2004). This may specifically happens for small ns and is for example the case for the Broder & Schutz example below. In these cases FIA cannot be used! Therefore, always check for consistency of the FIA penalty terms.

Once again: If one wants to compute the FIA, it is *absolutely necessary*, that the representation of the model via equations in the model file exactly maps on the structure of the binary MPT (see make.mpt.cf for more details).

Confidence intervals (CI) are based on the observed Hessian matrix produced by the symbolically derived function for the Hessian (i.e., the second derivative of the likelihood function). If it is based on a numerically estimated Hessian, a warning will be given.
For inequality restricted parameters, the CIs are computed using the parameter estimates' variance bounds (see Baldi & Batchelder, 2003; especially Equation 19). Note that these bounds represent the "worst case scenario" variances, and can lead to CIs outside parameter boundaries if the set of inequalities is large and/or the variances for the reparameterized model are large (Note that CIs for non-restricted parameters can be outside the parameter boundaries as well due to large variances).

To avoid local minima and instead find the maximum likelihood estimates it is useful to set n.optim > 1 with random starting values (see below). If n.optim > 1, the summary of the vector containing the Log-Likelihood values returned by each run of nlminb is added to the output (to check whether local minima were present). If the model is rather big, n.optim > 1 can be slow.

Multicore fitting is achieved via the snowfall package and needs to be initialized via sfInit. As initialization needs some time, you can either initialize multicore facilities yourself using sfInit() and setting the sfInit argument to FALSE (the default) or let MPTinR initialize multicore facilities by setting the sfInit argument to TRUE. The former is recommended as initializing snowfall takes some time and only needs to be done once if you run fit.mpt multiple times. If there are any problems with multicore fitting, first try to initialize snowfall outside MPTinR (e.g., sfInit(parallel=TRUE, cpus=2)). If this does not work, the problem is not related to MPTinR but to

snowfall (for support and references visit: https://www.imbi.uni-freiburg.de/parallel/). Note that you should *close* snowfall via `sfStop()` after using MPTinR.

The fitting/optimization is achieved via `nlminb` (Fox, Hall, & Schryer, 1978) a Newton based algorithm using the analytically derived gradient. In some cases (e.g., in case of empty cells) `nlminb` will not converge successfully in which `fit.mpt` will retry fitting using a numerically estimated gradient (with warning).

`fit.mpt` is just a comfortable wrapper around the workhorse `fit.mptinr`. `fit.mpt` produces the appropriate objective function, gradient function, hessian function, and prediction function that are handed over to `fit.mptinr` (functions are produced by symbolical derivation, see `D`). A function similar to `fit.mpt` is `fit.model` which has the additional arguments `lower.bound` and `upper.bound` allowing to fit other models than just MPTs and the possibility to indicate whether or not to use the analytically derived gradient or hessian for fitting (here this is automatically handled). Note that for MPTs (where upper and lower bounds of parameters are set to 0 and 1, respectively) `fit.mpt` is probably faster as the objective function is slightly faster (i.e., more optimized). However, for datasets with many empty cells trying `fit.model` with or without gradient or hessian can be worth a try.

Note that `fit.mptinr` can fit models with arbitrary (i.e., custom) objective functions.

The old version of this function using `optim`'s L-BFGS-B algorithm is `fit.mpt.old`.

Value

For individual fits (i.e., `data` is a `vector`) a `list` containing one or more of the following components from the best fitting model:

`goodness.of.fit`
: A `data.frame` containing the goodness of fit values for the model. `Log.Likelihood` is the Log-Likelihood value. `G.Squared`, `df`, and `p.value` are the G^2 goodness of fit statistic.

`information.criteria`
: A `data.frame` containing model information criteria based on the G^2 value. The FIA values(s) are presented if `fia` is not `NULL`.

`model.info`
: A `data.frame` containing other information about the model. If the rank of the Fisher matrix (`rank.fisher`) *does not* correspond to the number of parameters in the model (`n.parameters`) this indicates a serious issue with the identifiability of the model. A common reason is that one of the parameter estimates lies on the bound of the parameter space (i.e., 0 or 1).

`parameters`
: A data.frame containing the parameter estimates and corresponding confidence intervals. If a restriction file was present, the restricted parameters are marked.

`data`
: A `list` of two matrices; the first one (`observed`) contains the entered data, the second one (`predicted`) contains the predicted values.

For multi-dataset fits (i.e., `data` is a `matrix` or `data.frame`) a `list` with similar elements, but the following differences:
The first elements, `goodness.of.fit`, `information.criteria`, and `model.info`, contain the same information as for individual fits, but each are `lists` with three elements containing the respective values for: each individual in the list element `individual`, the sum of the individual values in the list element `sum`, and the values corresponding to the fit for the aggregated data in the list element

aggregated.
parameters is a list containing:

individual	A 3-dimensional array containing the parameter estimates ([,1,]), confidence intervals [,2:3,], and, if restrictions not NULL, column 4 [,4,] is 0 for non-restricted parameters, 1 for equality restricted parameters, and 2 for inequality restricted parameters. The first dimension refers to the parameters, the second to the information on each parameter, and the third to the individual/dataset.
mean	A data.frame with the mean parameter estimates from the individual estimates. No confidence intervals can be provided for these values.
aggregated	A data.frame containing the parameter estimates and corresponding confidence intervals for the aggregated data. If a restriction file was present, the restricted parameters are marked.

The element data contains two matrices, one with the observed, and one with the predicted data (or is a list containing lists with individual and aggregated observed and predicted data).

If n.optim > 1, the summary of the vector (matrix for multi-individual fit) containing the Log-Likelihood values returned by each run of optim is added to the output: fitting.runs

When output == "full" the list contains the additional items:

optim.runs	A list (or list of lists for multiple datasets) containing the outputs from all runs by nlminb (including those runs produced when fitting did not converge)
best.fits	A list (or list of lists for multiple datasets) containing the outputs from the runs by nlminb that had the lowest likelihood (i.e., the successful runs)
hessian	A list containing the Hessian matrix or matrices of the final parameter estimates.

Note

Warnings may relate to the optimization routine (e.g., Optimization routine [...] did not converge successfully).
In these cases it is recommended to rerun fit.mpt to check if the results are stable.

Note

All (model or restriction) files should end with an empty line, otherwise a warning will be shown.

The likelihood returned does not include the factorial constants of the multinomial probability-mass functions.

Author(s)

Henrik Singmann and David Kellen with help from Karl Christoph Klauer.

References

Baldi, P. & Batchelder, W. H. (2003). Bounds on variances of estimators for multinomial processing tree models. *Journal of Mathematical Psychology*, 47, 467-470.

Broeder, A., & Schuetz, J. (2009). Recognition ROCs are curvilinear-or are they? On premature arguments against the two-high-threshold model of recognition. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 35(3), 587. doi:10.1037/a0015279

Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing*, 16, 1190-1208.

Fox, P. A., Hall, A. P., & Schryer, N. L. (1978). The PORT Mathematical Subroutine Library. *CM Trans. Math. Softw.*, 4, 104-126. doi:10.1145/355780.355783

Knapp, B. R., & Batchelder, W. H. (2004). Representing parametric order constraints in multi-trial applications of multinomial processing tree models. *Journal of Mathematical Psychology*, 48, 215-229.

Moshagen, M. (2010). multiTree: A computer program for the analysis of multinomial processing tree models. *Behavior Research Methods*, 42, 42-54.

Navarro, D. J. (2004). A Note on the Applied Use of MDL Approximations. *Neural Computation*, 16(9), 1763-1768.

Purdy, B. P., & Batchelder, W. H. (2009). A context-free language for binary multinomial processing tree models. *Journal of Mathematical Psychology*, 53, 547-561.

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339.

Stahl, C. & Klauer, K. C. (2007). HMMTree: A computer program for latent-class hierarchical multinomial processing tree models. *Behavior Research Methods*, 39, 267- 273.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010a). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010b). On the minimum description length complexity of multinomial processing trees. *Journal of Mathematical Psychology*, 54, 291-303.

See Also

`check.mpt` for a function that can help in constructing models.

`select.mpt` for the function that performs model selection on results from `fit.mpt`.

`fit.model` for a similar wrapper for which you can specify upper and lower bounds of parameters (and whether or not `nlminb` uses the symbolically derived gradient and hessian)

`fit.mptinr` is the workhorse with which you can also fit your own objective functions.

Examples

```
# The first example fits the MPT model presented in Riefer and Batchelder (1988, Figure 1)
# to the data presented in Riefer and Batchelder (1988, Table 1)
# Note that Riefer and Batchelder (1988, pp. 328) did some hypotheses tests not replicated here.
# Instead, we use each condition (i.e., row in Table 1) as a different dataset.

# load the data
data(rb.fig1.data, package = "MPTinR")

#get the character string with the position of the model:
model1 <- system.file("extdata", "rb.fig1.model", package = "MPTinR")
model1.eqn <- system.file("extdata", "rb.fig1.model.eqn", package = "MPTinR")
```

```
# just fit the first dataset:
fit.mpt(rb.fig1.data[1,], model1, n.optim = 1)
fit.model(rb.fig1.data[1,], model1, n.optim = 1)

#fit all datasets:
fit.mpt(rb.fig1.data, model1, n.optim = 1)
fit.model(rb.fig1.data, model1, n.optim = 1)

#fit all datasets using the .EQN model file:
fit.mpt(rb.fig1.data, model1.eqn, n.optim = 1)

#fit using a textConnection (i.e., you can specify the model in your script/code):
model1.txt <- "p * q * r
p * q * (1-r)
p * (1-q) * r
p * (1-q) * (1-r) + (1-p)"
fit.mpt(rb.fig1.data, textConnection(model1.txt), n.optim = 1)



# The second example fits the MPT model presented in Riefer and Batchelder (1988, Figure 2)
# to the data presented in Riefer and Batchelder (1988, Table 3)
# First, the model without restrictions is fitted: ref.model
# Next, the model with all r set equal is fitted: r.equal
# Then, the model with all c set equal is fitted: c.equal
# Finally, the inferential tests reported by Riefer & Batchelder, (1988, p. 332) are executed.

# get the data
data(rb.fig2.data, package = "MPTinR")

# positions of model and restriction files:
model2 <- system.file("extdata", "rb.fig2.model", package = "MPTinR")
model2r.r.eq <- system.file("extdata", "rb.fig2.r.equal", package = "MPTinR")
model2r.c.eq <- system.file("extdata", "rb.fig2.c.equal", package = "MPTinR")

# The full (i.e., unconstrained) model
(ref.model <- fit.mpt(rb.fig2.data, model2))

# All r equal
(r.equal <- fit.mpt(rb.fig2.data, model2, model2r.r.eq))

# All c equal
(c.equal <- fit.mpt(rb.fig2.data, model2, model2r.c.eq))

# is setting all r equal a good idea?
(g.sq.r.equal <- r.equal[["goodness.of.fit"]][["G.Squared"]] -
ref.model[["goodness.of.fit"]][["G.Squared"]])
(df.r.equal <- r.equal[["goodness.of.fit"]][["df"]] -
ref.model[["goodness.of.fit"]][["df"]])
(p.value.r.equal <- pchisq(g.sq.r.equal, df.r.equal , lower.tail = FALSE))

# is setting all c equal a good idea?
```

```
(g.sq.c.equal <- c.equal[["goodness.of.fit"]][["G.Squared"]] -
ref.model[["goodness.of.fit"]][["G.Squared"]])
(df.c.equal <- c.equal[["goodness.of.fit"]][["df"]] -
ref.model[["goodness.of.fit"]][["df"]])
(p.value.c.equal <- pchisq(g.sq.c.equal, df.c.equal , lower.tail = FALSE))

# You can specify restrictions also via a list instead of an external file:
# All r equal
r.equal.2 <- fit.mpt(rb.fig2.data, model2, list("r0 = r1 = r2= r3 = r4"), n.optim = 5)
all.equal(r.equal, r.equal.2)

# All c equal
c.equal.2 <- fit.mpt(rb.fig2.data, model2, list("c0 = c1 = c2 = c3= c4"))
all.equal(c.equal, c.equal.2)


## Not run:

# Example from Broder & Schutz (2009)
# We fit the data from the 40 individuals from their Experiment 3
# We fit three different models:
# 1. Their 2HTM model: br.2htm
# 2. A restricted 2HTM model with Dn = Do: br.2htm.res
# 3. A 1HTM model (i.e., Dn = 0): br.1htm
# We fit the models with, as well as without, applied inequality restrictions (see Details)
# that is, for some models (.ineq) we impose: G1 < G2 < G3 < G4 < G5
# As will be apparent, the inequality restrictions do not hold for all individuals.
# Finally, we compute the FIA for all models, taking inequalities into account.

data(d.broeder, package = "MPTinR")
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
r.2htm <- system.file("extdata", "broeder.2htm.restr", package = "MPTinR")
r.1htm <- system.file("extdata", "broeder.1htm.restr", package = "MPTinR")
i.2htm <- system.file("extdata", "broeder.2htm.ineq", package = "MPTinR")
ir.2htm <- system.file("extdata", "broeder.2htm.restr.ineq", package = "MPTinR")
ir.1htm <- system.file("extdata", "broeder.1htm.restr.ineq", package = "MPTinR")

# fit the original 2HTM
br.2htm <- fit.mpt(d.broeder, m.2htm)
br.2htm.ineq <- fit.mpt(d.broeder, m.2htm, i.2htm)

# do the inequalities hold for all participants?
br.2htm.ineq[["parameters"]][["individual"]][,"estimates",]
br.2htm[["parameters"]][["individual"]][,"estimates",]
# See the difference between forced and non-forced inequality restrictions:
round(br.2htm[["parameters"]][["individual"]][,"estimates",] -
br.2htm.ineq[["parameters"]][["individual"]][,"estimates",],2)

# The same for the other two models
# The restricted 2HTM
br.2htm.res <- fit.mpt(d.broeder, m.2htm, r.2htm)
br.2htm.res.ineq <- fit.mpt(d.broeder, m.2htm, ir.2htm)
round(br.2htm.res[["parameters"]][["individual"]][,"estimates",] -
```

```
br.2htm.res.ineq[["parameters"]][["individual"]][,"estimates",],2)
# The 1HTM
br.1htm <- fit.mpt(d.broeder, m.2htm, r.1htm)
br.1htm.ineq <- fit.mpt(d.broeder, m.2htm, ir.1htm)
round(br.2htm.res[["parameters"]][["individual"]][,"estimates",] -
br.2htm.res.ineq[["parameters"]][["individual"]][,"estimates",],2)

# identical to the last fit of the 1HTM (using a list as restriction):
br.1htm.ineq.list <- fit.mpt(d.broeder, m.2htm, list("G1 < G2 < G3 < G4 < G5", "Dn = 0"))
all.equal(br.1htm.ineq, br.1htm.ineq.list)  # TRUE

# These results show that inequality restrictions do not hold for all datasets.
# (It would look differently if we excluded critical cases,
# i.e., 2, 6, 7, 10, 18, 21, 25, 29, 32, 34, 35, 37, 38)
# Therefore, we get the FIA for the models as computed above

br.2htm.fia <- fit.mpt(d.broeder, m.2htm, fia = 200000)
br.2htm.ineq.fia <- fit.mpt(d.broeder, m.2htm, i.2htm, fia = 200000)
br.2htm.res.fia <- fit.mpt(d.broeder, m.2htm, r.2htm, fia = 200000 )
br.2htm.res.ineq.fia <- fit.mpt(d.broeder, m.2htm, ir.2htm, fia = 200000)
br.1htm.fia <- fit.mpt(d.broeder, m.2htm, r.1htm, fia = 200000)
br.1htm.ineq.fia <- fit.mpt(d.broeder, m.2htm, ir.1htm, fia = 200000)

# Model selection using the FIA
(br.select <- select.mpt(list(br.2htm.fia, br.2htm.ineq.fia, br.2htm.res.fia,
                              br.2htm.res.ineq.fia, br.1htm.fia, br.1htm.ineq.fia)))

# The same results, ordered by FIA
br.select[order(br.select[,"delta.FIA.sum"]),]

# Note that FIA for individual data (.sum) is not consistent (i.e., the penalty
# for the nested model br.1htm.ineq.fia is not really smaller than the penalty
# for the superordinate model br.2htm.ineq.fia).
# Hence, one should use the aggregated data for this analysis (not shown here)!

# Compare this with the model selection not using FIA:
select.mpt(list(br.2htm, br.2htm.ineq, br.2htm.res, br.2htm.res.ineq, br.1htm, br.1htm.ineq))

# Only use the aggregated data:
d.broeder.agg <- colSums(d.broeder)
br.2htm.agg <- fit.mpt(d.broeder.agg, m.2htm)
br.2htm.res.agg <- fit.mpt(d.broeder.agg, m.2htm, r.2htm)
br.1htm.agg <- fit.mpt(d.broeder.agg, m.2htm, r.1htm)

select.mpt(list(br.2htm.agg, br.2htm.res.agg, br.1htm.agg), output = "full")


# compare speed of no multicore versus multicore for multiple datasets:

require(snowfall)
# change number of CPUs if more are available
nCPU = 2
sfInit( parallel=TRUE, cpus=nCPU, type = "SOCK" )
```

```
# NO multicore
system.time(fit.mpt(d.broeder, m.2htm))

# multicore:
system.time(fit.mpt(d.broeder, m.2htm, multicore = "individual"))

sfStop()

## End(Not run)
```

`fit.mpt.old` *Function to fit MPT models (old)*

Description

`fit.mpt.old` function fits *binary* multinomial processing tree models (MPT models; e.g., Riefer & Batchelder, 1988). However, this function is an old version using the L-BFGS-B optimization routine. See `fit.mpt` for the new version.

Usage

```
fit.mpt.old(
data,
model.filename,
restrictions.filename = NULL,
n.optim = 5,
fia = NULL,
ci = 95,
starting.values = NULL,
output = c("standard", "fia", "full"),
reparam.ineq = TRUE,
sort.param = TRUE,
model.type = c("easy", "eqn", "eqn2"),
multicore = c("none", "individual", "n.optim"), sfInit = FALSE, nCPU = 2
)
```

Arguments

`data`	Either a *numeric* `vector` for individual fit or a *numeric* `matrix` or `data.frame` for multi-individual fit. The data on each position (column for multi-individual fit) must correspond to the respective line in the model file. Fitting for multiple individuals can be parallelized via `multicore`.
`model.filename`	A character `vector` specifying the location and name of the model file.

`restrictions.filename`
: NULL or a character vector or a list of characters. The default is NULL which corresponds to no restrictions. A character vector specifies the location or name of the restrictions file. A list of characters contains the restrictions directly.

`n.optim`
: Number of optimization runs. Can be parallelized via multicore. Default is 5. If the number is high, fitting can take long for *large* models.

`fia`
: Number of random samples to be drawn in the Monte Carlo algorithm to estimate the Fisher Information Approximation (FIA), a minimum description length based measure of model complexity (see Wu, Myung & Batchelder, 2010). The default is NULL which corresponds to no computation of the FIA. Reasonable values (e.g., > 200000) can lead to long computation times (minutes to hours) depending on the size of the model. See Details.

`ci`
: A scalar corresponding to the size of the confidence intervals for the parameter estimates. Default is 95 which corresponds to 95% confidence intervals.

`starting.values`
: A vector or NULL. If NULL (the default), starting values for parameters are randomly drawn from a uniform distribution with the interval (0.05 - 0.95). If length(starting.values)==2, starting values are randomly drawn from a uniform distribution with the interval starting.values[1] - starting.values[2]). If length(starting.values) matches the number of parameters in the model, starting.values will be used as the starting values for fitting and n.optim will be set to 1. See Details.

`output`
: If "fia", fit.mpt will additionally return the results from get.mpt.fia (if calculated fia not equal NULL). If "full" fit.mpt will additionally return the results from get.mpt.fia and the output of optim.

`reparam.ineq`
: Logical. Indicates whether or not inequality restrictions (when present in the model file) should be enforced while fitting the model. If TRUE (the default) inequality restricted parameters will be reparameterized, if FALSE not. See Details.

`sort.param`
: Logical. If TRUE, parameters are alphabetically sorted in the parameter table. If FALSE, the first parameters in the parameter table are the non-restricted ones, followed by the restricted parameters. Default is TRUE.

`model.type`
: Character vector specifying whether the model file is formatted in the easy way ("easy"; i.e., each line represents all branches corresponding to a response category) or the traditional EQN syntax ("eqn" or "eqn2"; see Details and e.g., Stahl & Klauer, 2007). If model.filename ends with .eqn or .EQN, model.type is automatically set to "eqn". Default is "easy".

`multicore`
: Character vector. If not "none", uses snowfall for parallelization (which needs to be installed separately via install.packages(snowfall)). If "individual", parallelizes the optimization for each individual (i.e., data needs to be a matrix or data.frame). If "n.optim", parallelizes the n.optim optimization runs. Default is "none" which corresponds to no parallelization. Note that you need to initialize snowfall in default settings. See sfInit and Details.

`sfInit`
: Logical. Relevant if multicore is not "none". If TRUE, fit.mpt will initialize and close the multicore support. If FALSE, (the default) assumes that sfInit() was initialized before. See Details.

nCPU Scalar. Only relevant if multicore is not "none" and sfInit is TRUE. Number of CPUs used by snowfall. Default is 2.

Details

There is a new version of this function using nlminb and the analytically derived gradient and hessian. See fit.mpt. We recommend using the new version fit.mpt, only use this version if you are sure on what to do.

The model file is either of the easy format or the "classical" EQN format (see below).
In the easy format (the default) the model file contains all trees of the model. Trees are separated by at least one empty line. Everything to the right of a hash (#) is ignored (this behavior is new since version 0.9.2). Lines starting with a # are treated as empty. Each line in each tree corresponds to all branches of this tree (concatenated by a +) that correspond to one of the possible response categories. The position of each line must correspond to the position of this response category in the data object (for multi-individual fit to the respective column).

The difference between both types of EQN format ("eqn" or"eqn2") is the way the first line of the model file is treated. If model.file is set to "eqn", MPTinR will ignore the first line of the model file and will read the rest of the file (as does multiTree; Moshagen, 2010). If model.file is set to "eqn2" MPTinR will only read as many lines as indicated in the first line of the EQN model file (as does e.g., HMMTree; Stahl & Klauer, 2007). As default fit.mpt expects the easy format, but if the filename ends with .eqn or .EQN and model.type is "easy", model.type is set to "eqn"
For the EQN format consult one of the corresponding papers (see e.g., Moshagen, 2010; Stahl & Klauer, 2007). The positions in the data object (number of column for multi-individual fit) must correspond to the category number in the EQN file.

Note that names of parameters in the model file should not start with hank.. Variables with these names can lead to unforeseen problems as variables starting with these letters are internally used.

The restrictions file may contain (sequential) equality (i.e., =) and inequality (i.e., <) restrictions and must adhere to the following rules:
1. Inequalities first.
2. If a variable appears in an inequality restriction, it can not be on the LHS of any further restriction.
3. If a variable appears on RHS of an equality restriction, it can not appear on LHS of an equality restriction.
Note that only "<" is supported as inequality operator but not ">"!
Examples of restrictions are (the following could all appear in one restrictions file):

```
D1 < D2 < D3
D4 = D3
B1 = B3 = 0.3333
X4 = X5 = D3
```

Restrictions file may contain comments (i.e., everything to the right of a # will be ignored; new behavior since version 0.9.2)

For equality restrictions, the equality restricted parameters are simply exchanged with their restrictions before the fitting.
For inequality restricted parameters, the model is reparameterized so that only the rightmost parameter of an inequality restriction remains the original parameter. Each instance of the other parameters in this restriction is replaced by the product of the rightmost parameter and dummy parameters (see Knapp & Batchelder, 2004). This procedure (which is equivalent to method A described in

Knapp & Batchelder, 2004) leads to an equivalent model (although the binary MPT structure is not apparent in the resulting equations).
To prohibit this reparameterization (i.e., if the inequality restrictions hold without reparameterization), you can set `reparam.ineq` to `FALSE`. This can be useful for obtaining the FIA (see examples in Wu, Myung, & Batchelder, 2010).

The fitting/optimization is achieved via `optim`'s L-BFGS-B method by Byrd et al. (1995) with random starting values. To avoid local minima it is useful to set `n.optim` > 1. If `n.optim` > 1, the `summary` of the vector containing the Log-Likelihood values returned by each run of `optim` is added to the output (to check whether local minima were present). If the model is rather big, `n.optim` > 1 can be slow.

To obtain a measure of the model's complexity beyond the number of parameters (and taking inequality restrictions into account), set `fia` to a (reasonably high) scalar integer (i.e., a number). Then, `fit.mpt` will obtain the Fisher information approximation (FIA), a minimum description based measure of model complexity, using the algorithm provided by Wu, Myung, & Batchelder (2010a, 2010b) ported from Matlab to R. When performing model-selection, this measure is superior to other methods such as the Akaike information criterion (AIC) or Bayesian information criterion (BIC) which basically only take the number of parameters into account.
To get the FIA, `fit.mpt.old` performs the following steps:
1. The representation of the model as equations is transformed into the string representation of the model in the context-free language of MPT models (L-BMPT; Purdy & Batchelder, 2009). For this step to be successful it is *absolutely necessary* that the equations representing the model perfectly map the tree structure of the MPT. That is, the model file is only allowed to contain parameters, their negations (e.g., `Dn` and `(1 - Dn)`) and the operators + and *, but nothing else. Simplifications of the equations will seriously distort this step. This step is achieved by `make.mpt.cf`.
2. The context free representation of the model is then fed into the MCMC function computing the FIA (the port of BMPTFIA provided by Wu, Myung & Batchelder (2010a), see `bmpt.fia`).
(Actually, both steps are achieved by a call to `get.mpt.fia`)

Once again: If one wants to compute the FIA, it is *absolutely necessary*, that the representation of the model via equations in the model file exactly maps on the structure of the binary MPT (see `make.mpt.cf` for more details).

Confidence intervals (CI) are based on the observed Hessian matrix returned by the minimization algorithm (`optim`).
For inequality restricted parameters, the CIs are computed using the parameter estimates' variance bounds (see Baldi & Batchelder, 2003; especially equation 19). Note that these bounds represent the "worst case scenario" variances, and can lead to CIs outside parameter boundaries if the set of inequalities is large and/or the variances for the reparameterized model are large (Note that CIs for non-restricted parameters can be outside the parameter boundaries as well due to large variances).

To set the starting values for the fitting process (e.g., to avoid local minima) one can set `starting.values` to a vector of length 2. Then, starting values are randomly drawn from a uniform distribution from `starting.values[1]` to `starting.values[2]`.
Furthermore, one can specify the starting values individually by supplying a vector with the same length as the number of parameters. Starting values must be ordered according to the alphabetical order of the parameters. Use `check.mpt` for a function that returns the alphabetical order of the parameters. If one specifies the starting values like that, `n.optim` will be set to 1 as all other values would not make any sense (the optimization routine will produce identical results with identical starting values).

Multicore fitting is achieved via the `snowfall` package and needs to be initialized via `sfInit`. As

initialization needs some time, you can either initialize multicore facilities yourself using `sfInit()` and setting the `sfInit` argument to `FALSE` (the default) or let MPTinR initialize multicore facilities by setting the `sfInit` argument to `TRUE`. The former is recommended as initializing `snowfall` takes some time and only needs to be done once if you run `fit.mpt.old` multiple times. If there are any problems with multicore fitting, first try to initialize `snowfall` outside MPTinR (e.g., `sfInit( parallel=TRUE, cpus=2 )`). If this does not work, the problem is not related to MPTinR but to snowfall (for support and references visit: https://www.imbi.uni-freiburg.de/parallel/). Note that you need to *close* snowfall via `sfStop()` after using MPTinR.

`fit.model()` is essentially a copy of `fit.mpt.old` that allows the user to specify the upper and lower bounds of the parameters. This function can be used to fit other models than MPT models that can be described in a model file. That is, the model file can contain any type of valid R expressions including R functions (potentially self-written) visible in the global environment (i.e., not only +, *, and - as operators). Currently `fit.model` should be viewed as experimental.

Note that `fit.model()` is usually slower than `fit.mpt.old` as there are some more checks in the critical function calculating the likelihood of the model.

The `lower.bound` and `upper.bound` needs to be of length 1 or equal to the number of `free` parameters. If length > 1, parameters are mapped to the bounds in alphabetic order of the parameters. Use `check.mpt` to obtain the alphabetical order of parameters for your model.

While it should be possible to specify equality or fixed restrictions it will probably lead to unforeseen consequences to specify inequality restrictions for non-MPT models.

Value

For individual fits (i.e., `data` is a `vector`) a `list` containing one or more of the following components from the best fitting model:

`goodness.of.fit`
: A `data.frame` containing the goodness of fit values for the model. `Log.Likelihood` is the Log-Likelihood value. `G.Squared`, `df`, and `p.value` are the G^2 goodness of fit statistic.

`information.criteria`
: A `data.frame` containing model information criteria based on the G^2 value. The FIA values(s) are presented if `fia` is not `NULL`.

`model.info`
: A `data.frame` containing other information about the model. If the rank of the Hessian matrix (`rank.hessian`) *does not* correspond to the number of parameters in the model (`n.parameters`) this indicates a serious issue with the identifiability of the model.

`parameters`
: A data.frame containing the parameter estimates and corresponding confidence intervals. If a restriction file was present, the restricted parameters are marked.

`data`
: A `list` of two matrices; the first one (`observed`) contains the entered data, the second one (`predicted`) contains the predicted values.

For multi-individual fits (i.e., `data` is a `matrix` or `data.frame`) a `list` with similar elements, but the following differences.
The first elements, `goodness.of.fit`, `information.criteria`, and `model.info`, contain the same information as for individual fits, but each are `lists` with three elements containing the respective

values for: each individual in the list element `individual`, the sum of the individual values in the list element `sum`, and the values corresponding to the fit for the aggregated data in the list element `aggregated`.
`parameters` is a list containing:

`individual`	A 3-dimensional array containing the parameter estimates ([,1,]), confidence intervals [,2:3,], and, if restrictions not `NULL`, column 4 [,4,] is 0 for non-restricted parameters, 1 for equality restricted parameters, and 2 for inequality restricted parameters. The first dimension refers to the parameters, the second to the information on each parameter, and the third to the individuals.
`mean`	A `data.frame` with the mean parameter estimates from the individual estimates. No confidence intervals can be provided for these values.
`aggregated`	A data.frame containing the parameter estimates and corresponding confidence intervals for the aggregated data. If a restriction file was present, the restricted parameters are marked.

The element `data` contains two matrices, one with the `observed`, and one with the `predicted` data.

If `n.optim` > 1, the `summary` of the vector (matrix for multi-individual fit) containing the Log-Likelihood values returned by each run of `optim` is added to the output.

When using R (>= 2.13.0) compiling `fit.mpt.old` using `compilers` `cmpfun` can significantly improve fitting time.

Note

There may be several warnings `fit.mpt.old` throws while fitting MPT models. Most of them are not problematic and related to matrix operations needed for confidence intervals. Examples:
`In sqrt(var.params) : NaNs produced`
`In sqrt(min(var.bound.tmp)) : NaNs produced`
These warnings are not critical.

Other warnings may relate to the optimization routine (e.g., `Optimization routine [...] did not converge successfully`).
In these cases it is recommended to rerun `fit.mpt.old` to check if the results are stable.

Note

All (model or restriction) files should end with an empty line, otherwise a warning will be shown.

Author(s)

Henrik Singmann and David Kellen with help from Karl Christoph Klauer and Fabian Hoelzenbein.

References

Baldi, P. & Batchelder, W. H. (2003). Bounds on variances of estimators for multinomial processing tree models. *Journal of Mathematical Psychology*, 47, 467-470.

Broeder, A., & Schuetz, J. (2009). Recognition ROCs are curvilinear-or are they? On premature arguments against the two-high-threshold model of recognition. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 35(3), 587. doi:10.1037/a0015279

Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM J. Scientific Computing*, 16, 1190-1208.

Knapp, B. R., & Batchelder, W. H. (2004). Representing parametric order constraints in multi-trial applications of multinomial processing tree models. *Journal of Mathematical Psychology*, 48, 215-229.

Moshagen, M. (2010). multiTree: A computer program for the analysis of multinomial processing tree models. *Behavior Research Methods*, 42, 42-54.

Purdy, B. P., & Batchelder, W. H. (2009). A context-free language for binary multinomial processing tree models. *Journal of Mathematical Psychology*, 53, 547-561.

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339.

Stahl, C. & Klauer, K. C. (2007). HMMTree: A computer program for latent-class hierarchical multinomial processing tree models. *Behavior Research Methods*, 39, 267- 273.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010a). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010b). On the minimum description length complexity of multinomial processing trees. *Journal of Mathematical Psychology*, 54, 291-303.

See Also

`fit.mpt` for the current and recommended function fot fitting MPTs

Examples

```
## Not run:

# The first example fits the MPT model presented in Riefer and Batchelder (1988, Figure 1)
# to the data presented in Riefer and Batchelder (1988, Table 1)
# Note that Riefer and Batchelder (1988, pp. 328) did some hypotheses tests, that are not done here.
# Rather, we use each condition (i.e., row in Table 1) as a different individual.
# We try to use n.optim = 1 here, but this can lead to local minima
# In general we recommend to set n.optim >= 5

# load the data
data(rb.fig1.data)

#get the character string with the position of the model:
model1 <- system.file("extdata", "rb.fig1.model", package = "MPTinR")
model1.eqn <- system.file("extdata", "rb.fig1.model.eqn", package = "MPTinR")

# just fit the first "individual":
fit.mpt.old(rb.fig1.data[1,], model1, n.optim = 1)

#fit all "individuals":
fit.mpt.old(rb.fig1.data, model1, n.optim = 1)

#fit all "individuals" using the .EQN model file:
fit.mpt.old(rb.fig1.data, model1.eqn, n.optim = 1)
```

```
# The second example fits the MPT model presented in Riefer and Batchelder (1988, Figure 2)
# to the data presented in Riefer and Batchelder (1988, Table 3)
# First, the model without restrictions is fitted: ref.model
# Next, the model with all r set equal is fitted: r.equal
# Then, the model with all c set equal is fitted: c.equal
# Finally, the inferential tests reported by Riefer & Batchelder, (1988, p. 332) are executed.
# Note, that n.optim = 10, because of frequent local minima.

# get the data
data(rb.fig2.data)

# positions of model and restriction files:
model2 <- system.file("extdata", "rb.fig2.model", package = "MPTinR")
model2r.r.eq <- system.file("extdata", "rb.fig2.r.equal", package = "MPTinR")
model2r.c.eq <- system.file("extdata", "rb.fig2.c.equal", package = "MPTinR")

# The full (i.e., unconstrained) model
(ref.model <- fit.mpt.old(rb.fig2.data, model2, n.optim = 10))

# All r equal
(r.equal <- fit.mpt.old(rb.fig2.data, model2, model2r.r.eq, n.optim = 10))

# All c equal
(c.equal <- fit.mpt.old(rb.fig2.data, model2, model2r.c.eq, n.optim = 10))

# is setting all r equal a good idea?
(g.sq.r.equal <- r.equal[["goodness.of.fit"]][["G.Squared"]] -
ref.model[["goodness.of.fit"]][["G.Squared"]])
(df.r.equal <- r.equal[["goodness.of.fit"]][["df"]] -
ref.model[["goodness.of.fit"]][["df"]])
(p.value.r.equal <- pchisq(g.sq.r.equal, df.r.equal , lower.tail = FALSE))

# is setting all c equal a good idea?
(g.sq.c.equal <- c.equal[["goodness.of.fit"]][["G.Squared"]] -
ref.model[["goodness.of.fit"]][["G.Squared"]])
(df.c.equal <- c.equal[["goodness.of.fit"]][["df"]] -
ref.model[["goodness.of.fit"]][["df"]])
(p.value.c.equal <- pchisq(g.sq.c.equal, df.c.equal , lower.tail = FALSE))


# Example from Broeder & Schuetz (2009)
# We fit the data from the 40 individuals from their Experiment 3
# We fit three different models:
# 1. Their 2HTM model: br.2htm
# 2. A restricted 2HTM model with Dn = Do: br.2htm.res
# 3. A 1HTM model (i.e., Dn = 0): br.1htm
# We fit the models with, as well as without, applied inequality restrictions (see Details)
# that is, for some models (.ineq) we impose: G1 < G2 < G3 < G4 < G5
# As will be apparent, the inequality restrictions do not hold for all individuals.
# Finally, we compute the FIA for all models, taking inequalities into account.
# Note: The following examples will take some time (> 1 hour).
```

```
data(d.broeder)
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
r.2htm <- system.file("extdata", "broeder.2htm.restr", package = "MPTinR")
r.1htm <- system.file("extdata", "broeder.1htm.restr", package = "MPTinR")
i.2htm <- system.file("extdata", "broeder.2htm.ineq", package = "MPTinR")
ir.2htm <- system.file("extdata", "broeder.2htm.restr.ineq", package = "MPTinR")
ir.1htm <- system.file("extdata", "broeder.1htm.restr.ineq", package = "MPTinR")

# fit the original 2HTM
br.2htm <- fit.mpt.old(d.broeder, m.2htm)
br.2htm.ineq <- fit.mpt.old(d.broeder, m.2htm, i.2htm)

# do the inequalities hold for all participants?
br.2htm.ineq[["parameters"]][["individual"]][,"estimates",]
br.2htm[["parameters"]][["individual"]][,"estimates",]
# See the difference between forced and non-forced inequality restrictions:
round(br.2htm[["parameters"]][["individual"]][,"estimates",] -
br.2htm.ineq[["parameters"]][["individual"]][,"estimates",],2)

# The same for the other two models
# The restricted 2HTM
br.2htm.res <- fit.mpt(d.broeder, m.2htm, r.2htm)
br.2htm.res.ineq <- fit.mpt(d.broeder, m.2htm, ir.2htm)
round(br.2htm.res[["parameters"]][["individual"]][,"estimates",] -
br.2htm.res.ineq[["parameters"]][["individual"]][,"estimates",],2)
# The 1HTM
br.1htm <- fit.mpt(d.broeder, m.2htm, r.1htm)
br.1htm.ineq <- fit.mpt(d.broeder, m.2htm, ir.1htm)
round(br.2htm.res[["parameters"]][["individual"]][,"estimates",] -
br.2htm.res.ineq[["parameters"]][["individual"]][,"estimates",],2)

# These results show that we cannot compute inequality constraints for the non inequality
#  imposed models (It would look differently if we excluded critical cases,
# i.e., 2, 6, 7, 10, 18, 21, 25, 29, 32, 34, 35, 37, 38)
# Therefore, we get the FIA for the models as computed above
# WARNING: The following part will take a long time!

br.2htm.fia <- fit.mpt.old(d.broeder, m.2htm, fia = 200000)
br.2htm.ineq.fia <- fit.mpt.old(d.broeder, m.2htm, i.2htm, fia = 200000)
br.2htm.res.fia <- fit.mpt.old(d.broeder, m.2htm, r.2htm, fia = 200000 )
br.2htm.res.ineq.fia <- fit.mpt.old(d.broeder, m.2htm, ir.2htm, fia = 200000)
br.1htm.fia <- fit.mpt.old(d.broeder, m.2htm, r.1htm, fia = 200000)
br.1htm.ineq.fia <- fit.mpt.old(d.broeder, m.2htm, ir.1htm, fia = 200000)

# Model selection using the FIA
(br.select <- select.mpt(list(orig.2htm = br.2htm.fia, orig.2htm.ineq = br.2htm.ineq.fia,
res.2htm = br.2htm.res.fia, res.2htm.ineq = br.2htm.res.ineq.fia,
orig.1htm = br.1htm.fia, orig.1htm.ineq = br.1htm.ineq.fia)))
# The same results, ordered by FIA
br.select[order(br.select[,"delta.FIA.sum"]),]

# Compare this with the model selection not using FIA:
```

```
select.mpt(list(orig.2htm = br.2htm, orig.2htm.ineq = br.2htm.ineq,
res.2htm = br.2htm.res, res.2htm.ineq = br.2htm.res.ineq,
orig.1htm = br.1htm, orig.1htm.ineq = br.1htm.ineq))



# compare speed of no multicore versus multicore for multiple optimization runs:

require(snowfall)
# change number of CPUs if more are available
nCPU = 2
sfInit( parallel=TRUE, cpus=nCPU, type = "SOCK" )

# NO multicore
system.time(fit.mpt.old(d.broeder, m.2htm))

# multicore:
system.time(fit.mpt.old(d.broeder, m.2htm, multicore = "n.optim"))

sfStop()

## End(Not run)
```

`fit.mptinr` *Fit cognitive models for categorical data using an objective function*

Description

Fitting function for package MPTinR. Can fit any model for categorical data specified in an objective function. Fitting can be enhanced with gradient and or Hessian. Predicted values will be added when a prediction function is present.

Usage

```
fit.mptinr(
data,
objective,
param.names,
categories.per.type,
gradient = NULL, use.gradient = TRUE,
hessian = NULL, use.hessian = FALSE,
prediction = NULL,
n.optim = 5,
fia.df = NULL,
ci = 95,
starting.values = NULL,
```

```
    lower.bound = 0,
    upper.bound = 1,
    output = c("standard", "fia", "full"),
    fit.aggregated = TRUE,
    sort.param = TRUE,
    show.messages = TRUE,
    use.restrictions = FALSE,
    orig.params = NULL,
    restrictions = NULL,
    multicore = c("none", "individual", "n.optim"), sfInit = FALSE, nCPU = 2,
    control = list(),
        numDeriv = TRUE,
    ...
    )
```

Arguments

`data`	Either a *numeric* `vector` for individual fit or a *numeric* `matrix` or `data.frame` for multi-dataset fit. The data on each position (column for multi-dataset fit) must correspond to the respective line in the model file. Fitting for multiple datasets can be parallelized via `multicore`.
`objective`	the objective function used for fitting. Needs to return a scalar likelihood value.
`param.names`	character vector giving the parameters present in the model. The order of this vector determines the order with which the output from the fitting routine is interpreted.
`categories.per.type`	numeric vector indicating how many response categories each item type has.
`gradient`	the gradient function used for fitting. Needs to return a vector of same length as `param.names`.
`use.gradient`	logical. indicating whether or not `gradient` should be used for fitting. Default is `TRUE`
`hessian`	the Hessian function used for fitting. Needs to return a matrix with `dim = c(length(param.names), length(param.names))`.
`use.hessian`	logical. indicating whether or not `hessian` should be used for fitting. Default is `FALSE`
`prediction`	the prediction function. Needs to return a vector of equal length as the response categories or data. Needs to return probabilities!
`n.optim`	Number of optimization runs. Can be parallelized via `multicore`. Default is 5.
`fia.df`	needed for handling MPTs with computation of FIA coming from `fit.mpt` or `fit.model`. Do not use.
`ci`	A scalar corresponding to the size of the confidence intervals for the parameter estimates. Default is 95 which corresponds to 95% confidence intervals.
`starting.values`	A `vector`, a `list`, or `NULL` (the default). If `NULL` starting values for parameters are randomly drawn from a uniform distribution with the interval `(0.1 - 0.9)`. See Details for the other options.

lower.bound	numeric scalar or vector. Can be used in fit.model to set the lower bounds of the parameter space. See Details.
upper.bound	numeric scalar or vector. Can be used in fit.model to set the upper bounds of the parameter space. See Details.
output	If "full" fit.mpt will additionally return the return values of nlminb and the Hessian matrices. (If "fia", fit.mpt will additionally return the results from get.mpt.fia (if fia not equal NULL).)
fit.aggregated	logical. Only relevant for multiple datasets (i.e., matrix or data.frame). Should the aggregated dataset (i.e., data summed over rows) be additionally fitted? Default (TRUE) fits the aggregated data.
sort.param	Logical. If TRUE, parameters are alphabetically sorted in the parameter table. If FALSE, the first parameters in the parameter table are the non-restricted ones, followed by the restricted parameters. Default is TRUE.
show.messages	Logical. If TRUE the time the fitting algorithms takes is printed to the console.
use.restrictions	needed for handling MPTs coming from fit.mpt. Do not use, unless you are sure what you are doing.
orig.params	needed for handling models coming from fit.mpt or fit.model. Do not use, unless you are sure what you are doing.
restrictions	needed for handling models coming from fit.mpt or fit.model. Do not use, unless you are sure what you are doing.
multicore	Character vector. If not "none", uses snowfall for parallelization (which needs to be installed separately via install.packages(snowfall)). If "individual", parallelizes the optimization for each individual (i.e., data needs to be a matrix or data.frame). If "n.optim", parallelizes the n.optim optimization runs. Default is "none" which corresponds to no parallelization. Note that you need to initialize snowfall in default settings. See sfInit and Details.
sfInit	Logical. Relevant if multicore is not "none". If TRUE, fit.mpt will initialize and close the multicore support. If FALSE, (the default) assumes that sfInit() was initialized before. See Details.
nCPU	Scalar. Only relevant if multicore is not "none" and sfInit is TRUE. Number of CPUs used by snowfall. Default is 2.
control	list containing control arguments passed on to nlminb. See there.
numDeriv	logical. Should the Hessian matrix of the maximum likelihood estimates be estimated numerically using numDeriv::hessian in case it cannot be estimated analytically? This can be extremely time and memory consuming for larger models. Default is TRUE.
...	arguments passed on to the objective function, the gradient function, the hessian function and the prediction function.

Details

This functions can be used to fit any model for categorical data that can be specified via a (objective) function (i.e., especially models that are not MPTs). For fitting MPTs or other similar models such as SDTs see fit.mpt or fit.model.

The only mandatory arguments are: `data`, `objective`, `param.names`, and `categories.per.type`. Adding a function calculating the `gradient` will usually significantly speed up the fitting. However, in extreme cases (i.e., many empty cells) using the gradient can interfere with finding the global minima. Adding the function computing the `hessian` matrix is usually only useful for obtaining the accurate confidence intervals (usually the numerically estimated Hessian matrix is equivalent unless there are many empty cells or parameters at the boundary).

The `objective` (and `gradient` and `hessian`) function need to take as the first argument a numerical vector of `length(param.names)` representing the parameters. The other mandatory arguments for these functions are:
`data`: A vector containing the data for the dataset being fitted.
`param.names`: The character vector containing the parameter names is handed over to the objective.
`n.params`: = `length(param.names)`. To speed up computation the number of parameters is also handed over to the objective on each iteration.
`tmp.env`: A `environment` (created with `new.env`). The objective function produced by `fit.mpt` assign the parameter values into this environment using the following statement:
`for (i in seq_len(n.params)) assign(param.names[i],Q[i], envir = tmp.env)`
Furthermore, `fit.mptinr` assigns the data points before fitting each dataset into `tmp.env` with the variables names `hank.data.x` where `x` is the ordinal number of that data point (i.e., position or column). In other words, you can use `tmp.env` to `eval` you model within this environment and access both parameters and data in it.
`lower.bound` and `upper.bound`: both `lower.bound` and `upper.bound` will be passed on to the user-supplied functions as when nlminb fits without `gradient` it can try to use parameter values outside the bounds. This can be controlled with these arguments isnide the objective function.

Furthermore, note that all arguments passed via `...` will be passed to `objective`, `gradient`, and `hessian`. And that these three functions need to take the same arguments. Furthermore `gradient` must return a vector as long as `param.names` and `hessian` must return a square matrix of order `length(param.names)`. See `nlminb` for (slightly) more info.

Usage of `gradient` and/or `hessian` can be controlled with `use.gradient` and `use.hessian`.

`prediction` is a function similar to `objective` with the difference that it should return a vector of length `sum(categories.per.type` giving the probabilities for each item type. This function needs to take the same arguments as `objective` with the only exception that it does not take `lower.bound` and `upper.bound` (but `...` is passed on to it).

Note that parameters names should not start with `hank.`.

To set the starting values for the fitting process (e.g., to avoid local minima) one can set `starting.values` to a vector of length 2 and `n.optim` > 1. Then, starting values are randomly drawn from a uniform distribution from `starting.values[1]` to `starting.values[2]`.

Alternatively, one can supply a list with two elements to `starting.values`. Both elements need to be either of length 1 or of length equal to the number of parameters (if both are of length 1, it is the same as if you supply a vector of length 2). For each parameter n (in alphabetical order), a starting value is randomly drawn from a uniform distribution `starting.values[[1]][n]` to `starting.values[[2]][n]` (if length is 1, this is the border for all parameters).

The least interesting option is to specify the starting values individually by supplying a vector with the same length as the number of parameters. Starting values must be ordered according to the alphabetical order of the parameter names. Use `check.mpt` for a function that returns the alphabetical order of the parameters. If one specifies the starting values like that, `n.optim` will be

set to 1 as all other values would not make any sense (the optimization routine will produce identical results with identical starting values).

Multicore fitting is achieved via the `snowfall` package and needs to be initialized via `sfInit`. As initialization needs some time, you can either initialize multicore facilities yourself using `sfInit()` and setting the `sfInit` argument to `FALSE` (the default) or let MPTinR initialize multicore facilities by setting the `sfInit` argument to `TRUE`. The former is recommended as initializing `snowfall` takes some time and only needs to be done once if you run `fit.mpt` multiple times. If there are any problems with multicore fitting, first try to initialize `snowfall` outside MPTinR (e.g., `sfInit( parallel=TRUE, cpus=2 )`). If this does not work, the problem is not related to MPTinR but to snowfall (for support and references visit: https://www.imbi.uni-freiburg.de/parallel/). Note that you need to *close* snowfall via `sfStop()` after using MPTinR.

Value

For individual fits (i.e., `data` is a `vector`) a `list` containing one or more of the following components from the best fitting model:

`goodness.of.fit`
: A `data.frame` containing the goodness of fit values for the model. `Log.Likelihood` is the Log-Likelihood value. `G.Squared`, `df`, and `p.value` are the G^2 goodness of fit statistic.

`information.criteria`
: A `data.frame` containing model information criteria based on the G^2 value. The FIA values(s) are presented if `fia` is not `NULL`.

`model.info`
: A `data.frame` containing other information about the model. If the rank of the Fisher matrix (`rank.fisher`) *does not* correspond to the number of parameters in the model (`n.parameters`) this indicates a serious issue with the identifiability of the model. A common reason is that one of the parameter estimates lies on the bound of the parameter space (i.e., 0 or 1).

`parameters`
: A data.frame containing the parameter estimates and corresponding confidence intervals. If a restriction file was present, the restricted parameters are marked.

`data`
: A `list` of two matrices; the first one (`observed`) contains the entered data, the second one (`predicted`) contains the predicted values.

For multi-dataset fits (i.e., `data` is a `matrix` or `data.frame`) a `list` with similar elements, but the following differences:
The first elements, `goodness.of.fit`, `information.criteria`, and `model.info`, contain the same information as for individual fits, but each are `lists` with three elements containing the respective values for: each individual in the list element `individual`, the sum of the individual values in the list element `sum`, and the values corresponding to the fit for the aggregated data in the list element `aggregated`.
`parameters` is a list containing:

`individual`
: A 3-dimensional array containing the parameter estimates ([,1,]), confidence intervals [,2:3,], and, if restrictions not `NULL`, column 4 [,4,] is 0 for non-restricted parameters, 1 for equality restricted parameters, and 2 for inequality restricted parameters. The first dimension refers to the parameters, the second to the information on each parameter, and the third to the individual/dataset.

mean	A data.frame with the mean parameter estimates from the individual estimates. No confidence intervals can be provided for these values.
aggregated	A data.frame containing the parameter estimates and corresponding confidence intervals for the aggregated data. If a restriction file was present, the restricted parameters are marked.

The element data contains two matrices, one with the observed, and one with the predicted data (or is a list containing lists with individual and aggregated observed and predicted data).

If n.optim > 1, the summary of the vector (matrix for multi-individual fit) containing the Log-Likelihood values returned by each run of optim is added to the output: fitting.runs

When output == "full" the list contains the additional items:

optim.runs	A list (or list of lists for multiple datasets) containing the outputs from all runs by nlminb (including those runs produced when fitting did not converge)
best.fits	A list (or list of lists for multiple datasets) containing the outputs from the runs by nlminb that had the lowest likelihood (i.e., the successful runs)
hessian	A list containing the Hessian matrix or matrices of the final parameter estimates.

Note

Warnings may relate to the optimization routine (e.g., Optimization routine [...] did not converge successfully).
In these cases it is recommended to rerun the model to check if the results are stable.

Note

All (model or restriction) files should end with an empty line, otherwise a warning will be shown.

Author(s)

Henrik Singmann and David Kellen.

References

Kellen, D., Klauer, K. C., & Singmann, H. (2012). On the Measurement of Criterion Noise in Signal Detection Theory: The Case of Recognition Memory. *Psychological Review*. doi:10.1037/a0027727

See Also

fit.model or fit.mpt for a function that can fit model represented in a model file.

Examples

```
## Not run:
# the example may occasionally fail due to a starting values - integration mismatch.

# Fit an SDT for a 4 alternative ranking task (Kellen, Klauer, & Singmann, 2012).
```

```
ranking.data <- structure(c(39, 80, 75, 35, 61, 54, 73, 52, 44, 63, 40, 48, 80,
49, 43, 80, 68, 53, 81, 60, 60, 65, 49, 58, 69, 75, 71, 47, 44,
85, 23, 9, 11, 21, 12, 21, 14, 20, 19, 15, 29, 13, 14, 15, 22,
11, 12, 16, 13, 20, 20, 9, 26, 19, 13, 9, 14, 15, 24, 9, 19,
7, 9, 26, 16, 14, 6, 17, 21, 14, 20, 18, 5, 19, 17, 5, 11, 21,
4, 9, 15, 17, 7, 17, 11, 11, 9, 19, 20, 3, 19, 4, 5, 18, 11,
11, 7, 11, 16, 8, 11, 21, 1, 17, 18, 4, 9, 10, 2, 11, 5, 9, 18,
6, 7, 5, 6, 19, 12, 3), .Dim = c(30L, 4L))

expSDTrank <- function(Q, param.names, n.params, tmp.env){

    e <- vector("numeric",4)

    mu <- Q[1]
    ss <- Q[2]

    G1<-function(x){
        ((pnorm(x)^3)*dnorm(x,mean=mu,sd=ss))
    }

    G2<-function(x){
        ((pnorm(x)^2)*dnorm(x,mean=mu,sd=ss)*(1-pnorm(x)))*3
    }

     G3<-function(x){
        (pnorm(x)*dnorm(x,mean=mu,sd=ss)*(1-pnorm(x))^2)*3
    }


    e[1] <- integrate(G1,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[2] <- integrate(G2,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[3] <- integrate(G3,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[4] <- 1-e[1]-e[2]-e[3]

    return(e)
}



SDTrank <- function(Q, data, param.names, n.params, tmp.env, lower.bound, upper.bound){

    e<-vector("numeric",4)

    mu <- Q[1]
    ss <- Q[2]

    G1<-function(x){
        ((pnorm(x)^3)*dnorm(x,mean=mu,sd=ss))
    }

    G2<-function(x){
        ((pnorm(x)^2)*dnorm(x,mean=mu,sd=ss)*(1-pnorm(x)))*3
    }
```

```
     G3<-function(x){
        (pnorm(x)*dnorm(x,mean=mu,sd=ss)*(1-pnorm(x))^2)*3
     }


    e[1] <- integrate(G1,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[2] <- integrate(G2,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[3] <- integrate(G3,-Inf,Inf,rel.tol = .Machine$double.eps^0.5)$value
    e[4] <- 1-e[1]-e[2]-e[3]

    LL <- -sum(data[data!=0]*log(e[data!=0]))
    return(LL)
}

fit.mptinr(ranking.data, SDTrank, c("mu", "sigma"), 4, prediction = expSDTrank,
lower.bound = c(0,0.1), upper.bound = Inf)

## End(Not run)
```

---

gen.data *Generate or bootstrap data and get predictions from a model specified in a model file (or connection).*

Description

gen.data generates random dataset(s) from given paramater values and model (specified via model file or textConnection) for paramteric bootstrap.
sample.data generates random dataset(s) from given data for nonparametric bootstrap.
gen.predictions generates response probabilities or predicted responses from given paramater values and model (specified via model file or textConnection).

Usage

```
gen.data(parameter.values, samples,
model.filename,
data = NULL, n.per.item.type = NULL,
restrictions.filename = NULL, model.type = c("easy", "eqn", "eqn2"),
reparam.ineq = TRUE, check.model = TRUE)

sample.data(data, samples,
model.filename = NULL, categories.per.type = NULL,
model.type = c("easy", "eqn", "eqn2"), check.model = TRUE)

gen.predictions(parameter.values,
model.filename,
restrictions.filename = NULL,
```

```
n.per.item.type = NULL,
model.type = c("easy", "eqn", "eqn2"),
reparam.ineq = TRUE, check.model = TRUE)
```

Arguments

parameter.values	vector of paramater values. Either named then order is irrelevant or unnamed then must follow the alphabetical order of paramaters (check.mpt returns the alphabetical order of paramater names).
samples	Number of random datasets to be generated from a given set of paramater values.
n.per.item.type	vector of length equal to number of item types (or trees) specifying how many item each item type has. Default is NULL. See Details.
data	data vector. See Details.
categories.per.type	numeric vector indicating how many response categories each item type has.
model.filename	A character vector specifying the location and name of the model file, pssoble a textConnection.
restrictions.filename	NULL or a character vector or a list of characters. The default is NULL which corresponds to no restrictions. A character vector specifies the location or name of the restrictions file. A list of characters contains the restrictions directly. See fit.mpt for Details and Examples.
model.type	Character vector specifying whether the model file is formatted in the easy way ("easy"; i.e., each line represents all branches corresponding to a response category) or the traditional EQN syntax ("eqn" or "eqn2"; see Details and e.g., Stahl & Klauer, 2007). If model.filename ends with .eqn or .EQN, model.type is automatically set to "eqn". Default is "easy".
reparam.ineq	Should inequality restrictions be applied (i.e., the model reparametrized)? Default is TRUE.
check.model	logical. Should model be chekced with random values whether or not the expected values sum to one per tree? Default is TRUE.

Details

gen.data and sample.data are basically wrapper for rmultinom (called multiple times, if there is more than one item type). The prob argument of rmultinom is obtained differently for the two functions. For gen.data it corresponds to the predicted response proportions as returned by get.predictions (which is actually called by gen.data). For sample.data it is the proprtion of responses for each item type.

gen.data needs to know how big the n for each item type is. This can either be specified via the data or the n.per.item.type argument (i.e., one of those needs to be non-NULL). See the examples.

sample.data needs to know which response categories correspond to each item type. This can either be specified via the model.filename or the categories.per.type argument (i.e., one of those needs to be non-NULL). See the examples.

Value

Either a vector or matrix containing the generated data (for `gen.data` and `sample.data`) or a vector containing the predictions (for `gen.predictions`).

Author(s)

Henrik Singmann and David Kellen

See Also

`fit.mpt` or `fit.model` for functions that will fit the generated data. Note that it is probably a very good idea to set `fit.aggregated = FALSE` when fitting larger sets of generated data.

Examples

```
#### using the model and data from Broeder & Schuetz:
data(d.broeder, package = "MPTinR")
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
m.sdt <- "pkg/MPTinR/inst/extdata/broeder.sdt.model"

m.sdt <- system.file("extdata", "broeder.sdt.model", package = "MPTinR")

# fit the 2HTM
br.2htm <- fit.mpt(colSums(d.broeder), m.2htm)

# fit the SDT model
br.sdt <- fit.model(colSums(d.broeder), m.sdt, lower.bound = c(rep(-Inf, 5), 0, 1),
upper.bound = Inf)

# get one random dataset using the paramater values obtained (i.e., parametric bootstrap)
# and the data argument.
gen.data(br.2htm[["parameters"]][,1], 1, m.2htm, data = colSums(d.broeder))

gen.data(br.sdt[["parameters"]][,1], 1, m.sdt, data = colSums(d.broeder))

# get one random dataset using the paramater values obtained (i.e., parametric bootstrap)
# and the n.per.item.type argument.
gen.data(br.2htm[["parameters"]][,1], 1, m.2htm,
n.per.item.type = c(240, 2160, 600, 1800, 1200, 1200, 1800, 600, 2160, 240))

gen.data(br.sdt[["parameters"]][,1], 1, m.sdt,
n.per.item.type = c(240, 2160, 600, 1800, 1200, 1200, 1800, 600, 2160, 240))

# sample one random dataset from the original data:
sample.data(colSums(d.broeder), 1, model.filename = m.2htm)
# above uses the model.filename argument

sample.data(colSums(d.broeder), 1, categories.per.type = rep(2,10))
# above uses the categories.per.type argument


# just get the predicted proportions:
```

```
predictions.mpt <- gen.predictions(br.2htm[["parameters"]][,1], m.2htm)
predictions.sdt <- gen.predictions(br.sdt[["parameters"]][,1], m.sdt)

# predicting using the proactive Inhibiton Model (Riefer & Batchelder, 1988, Figure 1)

model1 <- system.file("extdata", "rb.fig1.model", package = "MPTinR")

gen.predictions(c(r = 0.3, p = 1, q = 0.4944), model1)
gen.predictions(c(r = 0.3, p = 1, q = 0.4944), model1, n.per.item.type = 180)

# the order of parameters is reordered (i.e., not alphabetically)
# but as the vector is named, it does not matter!
# Compare with:
data(rb.fig1.data, package = "MPTinR")
fit.mpt(rb.fig1.data[1,], model1, n.optim = 1)
```

`get.mpt.fia` *Convenient function to get FIA for MPT*

Description

`get.mpt.fia` is a comfortable wrapper for the R-port of Wu, Myung, and Batchelder's (2010) BMPTFIA `bmpt.fia`. It takes data, a model file, and (optionally) a restrictions file, computes the context-free representation of this file and then feeds this into `bmpt.fia` which returns the FIA.

Usage

```
get.mpt.fia(data, model.filename, restrictions.filename = NULL, Sample = 2e+05,
model.type = c("easy", "eqn", "eqn2"), round.digit = 6,
multicore = FALSE, split = NULL, mConst = NULL)
```

Arguments

`data`	Same as in `fit.mpt`
`model.filename`	Same as in `fit.mpt`
`restrictions.filename`	Same as in `fit.mpt`
`Sample`	The number of random samples to be drawn in the Monte Carlo algorithm. Default is 200000.
`model.type`	Same as in `fit.mpt`
`round.digit`	scalar numeric indicating to which decimal the ratios between ns in trees should be rounded (for minimizing computations with differing ns, see Details)
`multicore`	Same as in `bmpt.fia`
`split`	Same as in `bmpt.fia`
`mConst`	A constant which is added in the Monte Carlo integration to avoid numerical underflows and is later subtracted (after appropriate transformation). Should be a power of 2 to avoid unnecessary numerical imprecision.

Details

This function is called from `fit.mpt` to obtain the FIA, but can also be called independently.

It performs the following steps:
1.) Equality restrictions (if present) are applied to the model.
2.) The representation of the model as equations is transformed to the string representation of the model into the context-free language of MPT models (L-BMPT; Purdy & Batchelder, 2009). For this step to be successful it is *absolutely necessary* that the equations representing the model perfectly map the tree structure of the MPT. That is, the model file is only allowed to contain parameters, their negations (e.g., `Dn` and `(1 - Dn)`) and the operators + and *, but nothing else. Simplifications of the equations will seriously distort this step. This step is achieved by calling `make.mpt.cf`.
Note that inequality restrictions are not included in this transformation.
3.) The context free representation of the model is then fed into the MCMC function computing the FIA (the port of BMPTFIA provided by Wu, Myung, & Batchelder, 2010; see `bmpt.fia`).
If inequality restrictions are present, these are specified in the call to `bmpt.fia`.

For multi-individual data sets (i.e., `data` is a `matrix` or `data.frame`), `get.mpt.fia` tries to minimize computation time. That is done by comparing the ratios of the number items between trees. To not run into problems related to floating point precision, these values are rounded to `round.digit`. Then, `get.mpt.fia` will only call `bmpt.fia` as many times as there are differing ratios. As a consequence, the final penalty factor for FIA (`CFIA`) is calculated by `get.mpt.fia`, *without providing confidence intervals for the penalty factor*.

Value

A `data.frame` containing the results as returned by `bmpt.fia`:

`CFIA`	The FIA complexity value of the model with the corresponding confidence interval `CI.l` (lower bound) and `CI.u` (upper bound).
`lnInt`	The log integral term in C_FIA (Wu, Myung, & Batchelder, 2010a; Equation 7) for models without inequality constraints. When inequality constraints are present, 'lnInt' does not take into account the change in the normalizing constant in the proposal distribution and must be adjusted with the output argument 'lnconst'. The corresponding confidence interval ranges from `CI.lnint.l` (lower bound) to `CI.lnint.u` (upper bound).
`lnconst`	When inequality constraints are present, `lnconst` serves as an adjustment of codelnInt. It estimates the logarithm of the proportion of parameter space [0,1]^S that satisfies those inequality constraints, and the log integral term is given by `lnInt+lnconst`. The next [two] output argument [CI.lnconst] give the Monte Carlo confidence interval of 'lnconst'. [.l = lower & .u = upper bound of the CI]

Author(s)

Henrik Singmann

References

Purdy, B. P., & Batchelder, W. H. (2009). A context-free language for binary multinomial processing tree models. *Journal of Mathematical Psychology*, 53, 547-561.

Wu, H., Myung, J.I., & Batchelder, W.H. (2010). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

See Also

calls `bmpt.fia`
is called by `fit.mpt`, the main function for fitting MPT models

Examples

```
# Get the FIA for the 40 datasets from Broeder & Schuetz (2009, Experiment 3)
# for the 2HTM model with inequality restrictions
# (Can take a while.)

data(d.broeder)
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
i.2htm <- system.file("extdata", "broeder.2htm.ineq", package = "MPTinR")

get.mpt.fia(d.broeder, m.2htm, Sample = 1000) # Way too little samples
get.mpt.fia(d.broeder, m.2htm, i.2htm, Sample = 1000)

## Not run:
# should produce very similar results:
get.mpt.fia(d.broeder, m.2htm, i.2htm)
get.mpt.fia(d.broeder, m.2htm, i.2htm, mConst = 2L^8)

## End(Not run)
```

`make.eqn` *Creates an EQN model file oir MDT data file*

Description

`make.eqn` takes a model file in the "easy" format and creates a model file in the EQN format. `make.mdt` takes a data vector and produces an .mdt data file.

Usage

```
make.eqn(model.filename, eqn.filename)
make.mdt(data, mdt.filename, index, prefix = "dataset")
```

Arguments

`model.filename`	A character `vector` specifying the location and name of the model file in the `easy` format.
`eqn.filename`	A character vector specifying the location and name of the target .eqn file.
`data`	A vector, matrix or data.frame containing an individual data set to write to a .mdt file
`mdt.filename`	character vector specifying name and location of mdt file to be written.
`index`	index or second word written to the header of the mdt file. Ignored if `data` is a matrix or data.frame.
`prefix`	first word or prefix written to the mdt file. Default is `dataset`

Details

eqn and mdt files are the usual files used for programs to fit MPTs. You can use these functions to compare the results of MPTinR with other prgrams such as HMMTree or multiTree.

Note that these function do not add the endings .eqn or .mdt to the filename.

Since the MPTinR 0.9.4 `make.mdt` writes a single mdt file from a matrix or data.frame separating the participants via ===.

Value

Nothing

Author(s)

Henrik Singmann

References

More information on the .eqn format in e.g.:
Stahl, C., & Klauer, K. C. (2007). HMMTree: A computer program for latent-class hierarchical multinomial processing tree models. *Behavior Research Methods*, 39, 267-273.

See Also

`fit.mpt` for the main function of MPTinR

make.mpt.cf *Functions to transform MPT models.*

Description

(Helper) functions that takes an MPT model file and transforms it into a representation in the context-free language of MPT models L-BMPT (Purdy & Batchelder, 2009) or takes the representation in LBMPT and returns the model equations.

Usage

```
make.mpt.cf(model.filename, restrictions.filename = NULL,
            model.type = c("easy", "eqn"), treewise = FALSE)

lbmpt.to.mpt(model.list, outfile = NULL, category.names= TRUE)
```

Arguments

model.filename	A character vector specifying the location and name of the model file.
restrictions.filename	NULL or a character vector or a list of characters. The default is NULL which corresponds to no restrictions. A character vector specifies the location or name of the restrictions file. A list of characters contains the restrictions directly. Inequality/order restrictions are silently ignored.
model.type	Character vector specifying whether the model file is formatted in the easy format ("easy"; i.e., each line represents all branches corresponding to a response categories) or the traditional EQN syntax ("eqn" or "eqn2"). See Details in fit.mpt.
treewise	logical. Should the model be concatenated to one tree before transforming to LBMPT? Default is FALSE.
model.list	A list of character vectors representing a model in LBMPT. Each element of the vector corresponds to either a parameter or category. Each list element corresponds to one tree. Can be returned from make.mpt.cf.
outfile	Name of the file the model equation should be saved in (in easy format). If NULL (the default) prints it to the console instead (stdout()).
category.names	logical. Should category names (e.g., "category 1") be printed at the end of each line?

Details

Purdy and Batchelder (2009) provide a new way of how binary multinomial processing tree (MPT) models can be represented, a context free language called L-BMPT. This function takes a model file that consists of the equations defining a model and returns a character vector representing this model in L-BMPT.

There are three important things to know about this function:

1. L-BMPT distinguishes between observable categories (C) and parameters (theta). As MPTinR allows parameters to have any name that is legal for a variable in R (with the only restriction that parameters should NOT start with `hank`) the L-BMPT representation of the model's parameters can also consist of any name that is a legal variable name in R. To distinguish parameters from categories, categories are represented as integers (i.e., numbers) (a number is not a legal variable name in R). Furthermore, as legal variable names may end with a number, concatenating parameters and categories into one string could lead to an ambiguous representation of the model. Therefore, the returned representation in L-BMPT is a character vector with each element representing either a parameter (any legal variable name in R) or a category (an integer).
Theta: Names that are legal variable names in R.
C: Integers.

2. If a model consists of more than n > 1 trees, this function per default concatenates the trees into a single binary MPT model by adding n-1 parameters (named `hank.join.x` with `x` be a integer starting at 1; see Wu, Myung & Batchelder, 2010). This can be turned off, by setting `treewise` to TRUE.

3. It is absolutely necessary that the representation of the model via equations in the model file exactly maps on the structure of the binary tree. In other words, equations in the model file can NOT be simplified in any way. The equations in the model file may only consist of the parameters and their negations (e.g. `d` and `(1-d)`). Simplifications and aggregations in the model file (e.g., from `u * (1-u) + u * (1-u)` to `2*u*(1-u)` will lead to erroneous results! Similarly, reparameterizations for inequality constraints (which can be done by `fit.mpt`) can not be represented in L-BMPT.

Value

`make.mpt.cf`: A character vector with each element representing either a parameter or a category (categories are represented by integer numbers). In case of multiple trees and `treewise = TRUE`, a list of such vectors.

`lbmpt.to.mpt`: Either prints the model to the screen or returns nothing and saves the model equations in the specified file.

Note

It is absolutely necessary that the model file exactly maps on the structure of the binary tree. See Details.

Author(s)

Henrik Singmann (make.mpt.cf)

Quentin Gronau and Franz Dietrich (lbmpt.to.mpt, using a function from Akhil S Bhel, LinearizeNestedList)

References

Purdy, B. P., & Batchelder, W. H. (2009). A context-free language for binary multinomial processing tree models. *Journal of Mathematical Psychology*, 53, 547-561.

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339.

Wu, H., Myung, J., I., & Batchelder, William, H. (2010). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

See Also

`get.mpt.fia` and `link{bmpt.fia}` for functions calling `make.mpt.cf` to obtain the FIA of a MPT model.

See `fit.mpt` for the main function of MPTinR which also calls `make.mpt.cf` for obtaining the FIA.

`prepare.mpt.fia` will provide the code needed for obtaining the Minimum Description Length of a MPT model using Matlab (Wu, Myung & Batchelder, 2010) and calls `make.mpt.cf` for obtaining the L-BMPT representation.

Examples

```
model2 <- system.file("extdata", "rb.fig2.model", package = "MPTinR")

make.mpt.cf(model2)

make.mpt.cf(model2, treewise = TRUE)

lbmpt.to.mpt(make.mpt.cf(model2, treewise = TRUE))
```

---

prediction.plot *Plot observed versus predicted values for categorical data.*

Description

Plot observed minus predicted responses from a cognitive model for categorical data fit with MPTinR. Values above 0 indicate that there are to many responses in that category compared to the predictions, values below 0 indicate that there are to little responses compared to the predictions.

Usage

```
prediction.plot(results, model.filename,
dataset = 1,
absolute = TRUE,
spacing = 2,
axis.labels = NULL,
ylim, model.type = c("easy", "eqn", "eqn2"),
args.plot = list(), args.rect = list(), args.box = list(), args.points = list(),
args.labels = list(), numbers = c("individual", "continuous"),
pos.numbers = c("plot", "axis"), args.numbers = list(), args.abline = list(), abline)
```

Arguments

results	list. Results from fit.mpt, fit.model, or fit.mptinr.
model.filename	Same as in fit.mpt.
dataset	integer scalar or "aggregated" defining which dataset to plot.
absolute	logical. Should absolute deviations (the default) or G^2 deviations be plotted. See Details.
spacing	The spacing between two trees in x-axis ticks.
axis.labels	The labels on the x-axis. Default is Tree 1 to codeTree n.
ylim	the ylim argument to plot. If missing taken from data.
model.type	Same as in fit.mpt.
args.plot	list. Further arguments to plot, see details.
args.rect	list. Further arguments to rect, see details.
args.box	list. Further arguments to box, see details.
args.points	list. Further arguments to points, see details.
args.labels	list. Further arguments to axis, see details.
numbers	character vector (using partial matching) or NULL indicating where/if to plot numbers. Possible values are "individual", "continuous", or NULL. "individual" will start with 1 for the first response category in each tree/item type. "continuous" will use consecutive numbering matching the column numbers/ position of the data, NULL will plot no numbers. The default plots "individual" numbers.
pos.numbers	character vector, indicating where to plot the points. Possible values are "plot" or "axis" (using partial matching).
args.numbers	list. Further arguments to either text (if pos.numbers = "plot") or axis (if pos.numbers = "axis"), see details.
args.abline	list. Further arguments to abline, see details.
abline	logical. Whether to print vertical line at the positions of each point. If missing is set to TRUE if pos.numbers = "axis".

Details

This function uses base graphics to produce the plots and calls the following functions in the order given to do so: plot (produces an empty plot with axes), rect (produces the shaded area for each tree/ item type), box (produces another box around the plot), possibly abline (produces the vertical lines for each point), points (adds the data points), and depending on the value of numbers and pos.numbers either text (adds the numbers in the plot) or axis (adds the numbers below the plot).

For all of those functions default values are set but can be changed using the corresponding argument. These argument must be a named list containing arguments to that function (see Examples). Default arguments are:

- args.plot: list(xlab = "", ylab = "", main = "")
- args.rect: list(col = "grey", border = "transparent", density = 30, angle = 45)
- args.box: nothing

- args.points: `list(pch = 1, cex = 2.25)`
- args.labels: either `list(line = -1)` or if plotting `numbers` at `"axis"`, `list(line = 1)`
- args.numbers (if `pos.numbers = "plot"`): `list(labels = as.character(numbers.pch), cex = 0.7)` (If you change `"labels"` you can define what to plot instead of numbers).
- args.numbers (if `pos.numbers = "axis"`): `list(labels = numbers.pch, cex.axis = 0.6, mgp = c(0,0.3,0))` (If you change `"labels"` you can define what to plot instead of numbers)
- args.abline: `list(col = "darkgrey")`.

If `absolute = TRUE` (the default) absolute deviations are plotted (i.e., observed - predicted from the model). If `absolute = FALSE` G^2 values are plotted which are computed for all predictions where data is non 0 with:

$$2 \times \text{data} \times (log(\text{data}) - log(\text{predictions}))$$

Value

Invoked for its side effects, but invisibly returns a `list` with the x and y positions for each point.

Note

Please report all problems.

Author(s)

Henrik Singmann. Thanks to David Kellen for discussion and ideas.

See Also

`fit.mpt`

Examples

```
## Not run:
#### using the model and data from Broeder & Schuetz:
data(d.broeder, package = "MPTinR")
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
m.sdt <- "pkg/MPTinR/inst/extdata/broeder.sdt.model"

m.sdt <- system.file("extdata", "broeder.sdt.model", package = "MPTinR")

# fit the 2HTM
br.2htm <- fit.mpt(d.broeder, m.2htm)

# graphical parameters
par(mfrow = c(2,2))
prediction.plot(br.2htm, m.2htm, 4)
prediction.plot(br.2htm, m.2htm, 4, ylim = c(-4, 4), numbers = NULL,
args.points = list(pch = 16, cex = 1.5))
prediction.plot(br.2htm, m.2htm, 4, ylim = c(-4, 4), args.plot = list(main = "Dataset 4 - A"),
abline = TRUE, numbers = "continuous")
prediction.plot(br.2htm, m.2htm, 4, ylim = c(-4, 4), args.plot = list(main = "Dataset 4 - B"),
```

```
pos.numbers = "axis", abline = TRUE,
args.numbers = list(mgp = c(3, 0.2, 0), cex.axis = 0.35),
args.points = list(pch = 4, cex = 1.5))
dev.off()

prediction.plot(br.2htm, m.2htm, "aggregated", axis.labels = unlist(lapply(c(10, 25, 50, 75, 90),
paste, c("o.o", "o.n"), sep = "")))

# fit the SDT
br.sdt <- fit.model(d.broeder, m.sdt, lower.bound = c(rep(-Inf, 5), 0, 1), upper.bound = Inf)

axis.labels <- unlist(lapply(c(10, 25, 50, 75, 90), paste, c("o.o", "o.n"), sep = ""))
# compare predictions for aggregated data:
par(mfrow = c(2,2))
prediction.plot(br.2htm, m.2htm, "aggregated", ylim = c(-30, 30),
args.plot = list(main = "MPT model - absolute"), axis.labels = axis.labels)
prediction.plot(br.sdt, m.2htm, "aggregated", ylim = c(-30, 30),
args.plot = list(main = "SDT model - absolute"), axis.labels = axis.labels)
prediction.plot(br.2htm, m.2htm, "aggregated", ylim = c(-60, 60),
args.plot = list(main = "MPT model - G.squared"), absolute = FALSE,
axis.labels = axis.labels, pos.numbers = "axis", args.points = list(pch = 8, cex = 1))
prediction.plot(br.sdt, m.2htm, "aggregated", ylim = c(-60, 60),
args.plot = list(main = "SDT model - G.squared"), absolute = FALSE,
axis.labels = axis.labels, pos.numbers = "axis", args.points = list(pch = 8, cex = 1))

# comparing absoulte and G-sqaured plot with zero counts in cell 2:
par(mfrow = c(2,2))
prediction.plot(br.2htm, m.2htm, 2, ylim = c(-1, 1),
args.plot = list(main = "MPT model - absolute"))
prediction.plot(br.sdt, m.2htm, 2, ylim = c(-1, 1),
args.plot = list(main = "SDT model - absolute"))
prediction.plot(br.2htm, m.2htm, 2, ylim = c(-2, 2),
args.plot = list(main = "MPT model - G.squared"), absolute = FALSE)
prediction.plot(br.sdt, m.2htm, 2, ylim = c(-2, 2),
args.plot = list(main = "SDT model - G.squared"), absolute = FALSE)

## End(Not run)
```

`prepare.mpt.fia` *Provides MATLAB command to get FIA*

Description

This function needs data and a model files and outputs the exact command needed to obtain the minimum description length measure for MPT models using the procedure by Wu, Myung, and Batchelder (2010) for MATLAB. It can be considered an extended wrapper for `make.mpt.cf`.

Usage

```
prepare.mpt.fia(data, model.filename, restrictions.filename = NULL,
outfile = "clipboard", Sample = 2e+05, model.type = c("easy", "eqn", "eqn2"))
```

Arguments

`data`	Either a *numeric* `vector` for individual fit or a *numeric* `matrix` or `data.frame` for multi-individual fit. The data on each position (column for multi individual fit) must correpsond to the relevant line in the model file.
`model.filename`	A character `vector` specifying the location and name of the model file.
`restrictions.filename`	`NULL` or a character `vector` specifying the location and name of the restrictions file. Default is `NULL` which corresponds to no restrictions.
`outfile`	A character vector specifying the name of the file where the MATLAB code is saved. Default is `"clipboard"` which will copy the output to the clipboard and will not write it to a file (Windows only). Actually, this parameter is directly passed to `writeLines` which interprets character vectors as filenames, so any other legal `connection` can be used.
`Sample`	Number of Monte Carlo samples to be used by the procedure of Wu, Myung, and Batchelder (2010). Default is 200.000.
`model.type`	Character vector specifying whether the model file is formatted in the `easy` format (`"easy"`; i.e., each line represents all branches corresponding to a response categories) or the traditional EQN syntax (`"eqn"` or `"eqn2"`). See Details in `fit.mpt`.

Details

This function uses `make.mpt.cf` to create the representation in the L-BMPT. Therefore, it is necessary that the representation of the model via equations in the model file exactly maps on the structure of the binary tree (see `make.mpt.cf` for more details).

Whereas `fit.mpt` can reparameterize MPT models for fitting inequality constraints, Wu, Myung, and Batchelder (2010) have used another method to deal with these issues that is also adopted here. Our function does not report a reparameterized version of the MPT model that satisfies the inequality contraints, but modifies the appropriate argument in the call to the function by Wu et al (2010).

Note that MATLAB needs the statistics toolbox to run the script by Wu, Myung, and Batchelder (2010).

Value

The most important value is the output to a file or clipboard (Windows only) of the MATLAB code to get the minimum description length. For multiple individuals multiple outputs are generated which only differ if the ns of the data differ. Furthermore, each argument is returned in a list:

`s`	The string representation of the model.
`parameters`	A list of the numbers representing the parameters.

param.codes	A vector describing which number corresponds to which parameter in the parameters vector.
category	The numbers representing the categories.
ineq	The matrix representing the inequality constraints.
n	The n of the data.
internal	The L-BMPT representation as returned by make.mpt.cf.

Author(s)

Henrik Singmann

References

Wu, H., Myung, J., I., & Batchelder, William, H. (2010). Minimum description length model selection of multinomial processing tree models. *Psychonomic Bulletin & Review*, 17, 275-286.

See Also

Since we ported the original BMPTFIA function by Wu, Myung, & Batchelder (2010) to R (bmpt.fia), this function is a little bit outdated. However, getting the FIA in Matlab is (still) faster than getting it in R.

See also get.mpt.fia which takes the same arguments but will then compute the FIA using the function provided by Wu et al. (2010) ported to R.

make.mpt.cf

Examples

```
## Not run:
# This example produces the code for the first example of how to use the
# function by Wu, Myung & Batchelder (2010, pp. 280):
# Value should be around 12.61 and 12.62

model.1htm <- system.file("extdata", "wmb.ex1.model", package = "MPTinR")
model.1htm.restr <- system.file("extdata", "wmb.ex1.restr", package = "MPTinR")

prepare.mpt.fia(c(250,0,0,250,0,0,500,0,0), model.1htm, model.1htm.restr)

## End(Not run)
```

---

rb.fig1.data *Data to be used for the examples of MPTinR.*

Description

Dataset 1 (fig1) is taken from Riefer & Batchelder (1988, Table 1) and contains multiple individuals.
Dataset 2 (fig2) is taken from Riefer & Batchelder (1988, Table 3).

Usage

```
data(rb.fig1.data)
data(rb.fig2.data)
```

Source

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339.

ROCs *Recognition memory ROCs used by Klauer & Kellen (2015)*

Description

Data of the meta-analysis on recognition memory ROC (receiver operating characteristic) curves reported in Klauer and Kellen (2015). In total there are 850 individual ROCs, 459 6-point ROCs and 391 8-point ROCs. Both data sets first report responses to old items and then to new item. For both item types the response categories are ordered from sure-new to sure-old. Please always cite the original authors when using this data.

Usage

```
data("roc6")
data("roc8")
```

Details

The source of each data set is given in the `exp` column of each data set. The `id` column gives a unique id for each data set. For the 6-point ROCs the first 12 columns contain the data, for the 8-point ROCs, the first 16 columns.

Note

Whenever using any of the data available here, please make sure to cite the original sources given in the following.

Source

The 6-point ROCs contains data from the following sources:

- `Dube_2012-P` and `Dube_2012-W`:
  Dube, C., & Rotello, C. M. (2012). Binary ROCs in perception and recognition memory are curved. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 38(1), 130-151. doi:10.1037/a0024957
- `heathcote_2006_e1` and `heathcote_2006_e2`:
  Heathcote, A., Ditton, E., & Mitchell, K. (2006). Word frequency and word likeness mirror effects in episodic recognition memory. `Memory & Cognition`, 34(4), 826-838. doi:10.3758/BF03193430

- `Jaeger_2013`:
  Jaeger, A., Cox, J. C., & Dobbins, I. G. (2012). Recognition confidence under violated and confirmed memory expectations. Journal of Experimental Psychology: General, 141(2), 282-301. doi:10.1037/a0025687
- `Koen_2010_pure`:
  Koen, J. D., & Yonelinas, A. P. (2010). Memory variability is due to the contribution of recollection and familiarity, not to encoding variability. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 36(6), 1536-1542. doi:10.1037/a0020448
- `Koen_2011`:
  Koen, J. D., & Yonelinas, A. P. (2011). From humans to rats and back again: Bridging the divide between human and animal studies of recognition memory with receiver operating characteristics. *Learning & Memory*, 18(8), 519-522. doi:10.1101/lm.2214511
- `Koen-2013_full` and `Koen-2013_immediate`:
  Koen, J. D., Aly, M., Wang, W.-C., & Yonelinas, A. P. (2013). Examining the causes of memory strength variability: Recollection, attention failure, or encoding variability? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 39(6), 1726-1741. doi:10.1037/a0033671
- `Pratte_2010`:
  Pratte, M. S., Rouder, J. N., & Morey, R. D. (2010). Separating mnemonic process from participant and item effects in the assessment of ROC asymmetries. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 36(1), 224-232. doi:10.1037/a0017682
- `Smith_2004`:
  Smith, D. G., & Duncan, M. J. J. (2004). Testing Theories of Recognition Memory by Predicting Performance Across Paradigms. *Journal of Experimental Psychology. Learning, Memory & Cognition*, 30(3), 615-625.

The 8-point ROCs contains data from the following sources:

- `Benjamin_2013`:
  Benjamin, A. S., Tullis, J. G., & Lee, J. H. (2013). Criterion Noise in Ratings-Based Recognition: Evidence From the Effects of Response Scale Length on Recognition Accuracy. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 39, 1601-1608. doi:10.1037/a0031849
- `Onyper_2010-Pics` and `Onyper_2010-Words`:
  Onyper, S. V., Zhang, Y. X., & Howard, M. W. (2010). Some-or-none recollection: Evidence from item and source memory. *Journal of Experimental Psychology: General*, 139(2), 341-364. doi:10.1037/a0018926

References

Klauer, K. C., & Kellen, D. (2015). The flexibility of models of recognition memory: The case of confidence ratings. *Journal of Mathematical Psychology*, 67, 8-25. doi:10.1016/j.jmp.2015.05.002

Examples

```
## Not run:
# This example shows only how to fit the 6-point ROCs
data("roc6")
```

```
# 2HTM (2-high threshold model)
htm <- "
(1-Do)*(1-g)*(1-gn1)*(1-gn2)
(1-Do)*(1-g)*(1-gn1)*gn2
(1-Do)*(1-g)*gn1
Do*(1-do1)*(1-do2)  + (1-Do)*g*go1
Do*do1 + (1-Do)*g*(1-go1)*go2
Do*(1-do1)*do2 + (1-Do)*g*(1-go1)*(1-go2)

Dn*(1-dn1)*dn2 + (1-Dn)*(1-g)*(1-gn1)*(1-gn2)
Dn*dn1 + (1-Dn)*(1-g)*(1-gn1)*gn2
Dn*(1-dn1)*(1-dn2) + (1-Dn)*(1-g)*gn1
(1-Dn)*g*go1
(1-Dn)*g*(1-go1)*go2
(1-Dn)*g*(1-go1)*(1-go2)
"

# full 2HTM is over-parametereized:
check.mpt(textConnection(htm))
# apply some symmetric response mapping restrictions for D and g:
check.mpt(textConnection(htm), list("dn2 = do2", "gn2 = go2"))



# UVSD (unequal variance signal detection model)
uvsd <- "
pnorm(cr1, mu, sigma)
pnorm(cr1+cr2, mu, sigma) - pnorm(cr1, mu, sigma)
pnorm(cr3+cr2+cr1, mu, sigma) - pnorm(cr2+cr1, mu, sigma)
pnorm(cr4+cr3+cr2+cr1, mu, sigma) - pnorm(cr3+cr2+cr1, mu, sigma)
pnorm(cr5+cr4+cr3+cr2+cr1, mu, sigma) - pnorm(cr4+cr3+cr2+cr1, mu, sigma)
1 - pnorm(cr5+cr4+cr3+cr2+cr1, mu, sigma)

pnorm(cr1)
pnorm(cr2+cr1) - pnorm(cr1)
pnorm(cr3+cr2+cr1) - pnorm(cr2+cr1)
pnorm(cr4+cr3+cr2+cr1) - pnorm(cr3+cr2+cr1)
pnorm(cr5+cr4+cr3+cr2+cr1) - pnorm(cr4+cr3+cr2+cr1)
1 - pnorm(cr5+cr4+cr3+cr2+cr1)
"

# confidence criteria are parameterized as increments:
check.mpt(textConnection(uvsd))
# cr1 = [-Inf, Inf]
# cr2, cr3, cr4, cr5 = [0, Inf]
# mu = [-Inf, Inf]
# sigma = [0, Inf]


# MSD (mixture signal detection model):
# NOTE: To follow CRAN rules restricting examples to a width of 100 characters,
# the following example is splitted into multiple strings concatenated by paste().
```

```
# To view the full model use: cat(msd)
msd <- paste(c("
l*(pnorm(cr1-mu))  +  (1 - l) * (pnorm(cr1-mu2))
l*(pnorm(cr1+cr2-mu) - pnorm(cr1-mu)) + (1 - l)*(pnorm(cr1+cr2-mu2)-pnorm(cr1-mu2))
l*(pnorm(cr1+cr2+cr3-mu)-pnorm(cr1+cr2-mu)) + (1-l)*(pnorm(cr1+cr2+cr3-mu2)-pnorm(cr1+cr2-mu2))
",
"l*(pnorm(cr1+cr2+cr3+cr4-mu) - pnorm(cr1+cr2+cr3-mu)) + ",
"(1 - l)*(pnorm(cr1+cr2+cr3+cr4-mu2)-pnorm(cr1+cr2+cr3-mu2))",
"
l*(pnorm(cr1+cr2+cr3+cr4+cr5-mu)-pnorm(cr1+cr2+cr3+cr4-mu)) + ",
"(1 - l)*(pnorm(cr1+cr2+cr3+cr4+cr5-mu2)-pnorm(cr1+cr2+cr3+cr4-mu2))",
"
l * (1-pnorm(cr1+cr2+cr3+cr4+cr5-mu))  +  (1 - l)*(1-pnorm(cr1+cr2+cr3+cr4+cr5-mu2))

pnorm(cr1)
pnorm(cr1+cr2) - pnorm(cr1)
pnorm(cr1+cr2+cr3) - pnorm(cr1+cr2)
pnorm(cr1+cr2+cr3+cr4) - pnorm(cr1+cr2+cr3)
pnorm(cr1+cr2+cr3+cr4+cr5) - pnorm(cr1+cr2+cr3+cr4)
1-pnorm(cr1+cr2+cr3+cr4+cr5)
"), collapse = "")
cat(msd)

# confidence criteria are again parameterized as increments:
check.mpt(textConnection(msd))
# cr1 = [-Inf, Inf]
# cr2, cr3, cr4, cr5 = [0, Inf]
# lambda = [0, 1]
# mu, mu2 = [-Inf, Inf]


# DPSD (dual-process signal detection model)
dpsd <- "
(1-R)*pnorm(cr1- mu)
(1-R)*(pnorm(cr1 + cr2 - mu) - pnorm(cr1 - mu))
(1-R)*(pnorm(cr1 + cr2 + cr3 - mu) - pnorm(cr1 + cr2 - mu))
(1-R)*(pnorm(cr1 + cr2 + cr3 + cr4 - mu) - pnorm(cr1 + cr2 + cr3 - mu))
(1-R)*(pnorm(cr1 + cr2 + cr3 + cr4 + cr5 - mu) - pnorm(cr1 + cr2 + cr3 + cr4 - mu))
R + (1-R)*(1 - pnorm(cr1 + cr2 + cr3 + cr4 + cr5 - mu))

pnorm(cr1)
pnorm(cr1 + cr2) - pnorm(cr1)
pnorm(cr1 + cr2 + cr3) - pnorm(cr1 + cr2)
pnorm(cr1 + cr2 + cr3 + cr4) - pnorm(cr1 + cr2 + cr3)
pnorm(cr1 + cr2 + cr3 + cr4 + cr5) - pnorm(cr1 + cr2 + cr3 + cr4)
1 - pnorm(cr1 + cr2 + cr3 + cr4 + cr5)
"

uvsd_fit <- fit.model(roc6[,1:12], textConnection(uvsd),
            lower.bound=c(-Inf, rep(0, 5), 0.001), upper.bound=Inf)

msd_fit <- fit.model(roc6[,1:12], textConnection(msd),
            lower.bound=c(-Inf, rep(0, 7)), upper.bound=c(rep(Inf, 5), 1, Inf, Inf))
```

```
dpsd_fit <- fit.model(roc6[,1:12], textConnection(dpsd),
            lower.bound=c(-Inf, rep(0, 6)), upper.bound=c(rep(Inf, 6), 1))

htm_fit <- fit.mpt(roc6[,1:12], textConnection(htm),
                   list("dn2 = do2", "gn2 = go2"))

select.mpt(list(uvsd_fit, dpsd_fit, msd_fit, htm_fit))
# Note that the AIC and BIC results do not adequately take model flexibility into account.
##      model n.parameters G.Squared.sum df.sum p.sum p.smaller.05
## 1 uvsd_fit            7      1820.568   1377     0           50
## 2 dpsd_fit            7      2074.188   1377     0           64
## 3  msd_fit            8      1345.595    918     0           51
## 4  htm_fit            9      1994.217    459     0          138
##   delta.AIC.sum wAIC.sum AIC.best delta.BIC.sum wBIC.sum BIC.best
## 1        0.0000        1      230        0.0000        1      273
## 2      253.6197        0      161      253.6197        0      183
## 3      443.0270        0       16     4996.8517        0        3
## 4     2009.6489        0       56    11117.2982        0        4

## End(Not run)
```

`select.mpt` *Model Selection with MPTinR*

Description

This function performs model selection for results produced by MPTinR's `fit.mpt`. It takes multiple results from `fit.mpt` as a list and returns a `data.frame` comparing the models using various model selection criteria (e.g., FIA) and AIC and BIC weights. For model selection of multi-dataset fits `select.mpt` will additionally count how often each model provided the best fit.

Usage

```
select.mpt(mpt.results, output = c("standard", "full"), round.digit = 6, dataset)
```

Arguments

`mpt.results`	A `list` containing results from `fit.mpt`.
`output`	`"standard"` or `"full"`. If `"full"` additionally returns original FIA, AIC, and BIC values, and, for multi-individual fits, compares the model-selection criteria for the aggregated data.
`round.digit`	Integer specifying to which decimal place the results should be rounded. Default is 6. Is also used for rounding FIA, AIC, and BIC values before counting the best fitting values per individual datasets.
`dataset`	Integer vector specifying whether or not to restrict the individual comparison top certain dataset(s). Aggregated results will not be displayed if this argument is present.

Details

`select.mpt` is the second major function of MPTinR, next to `fit.mpt`. It takes a list of results produced by `fit.mpt` and returns a `data.frame` comparing the models using the information criteria obtained by `fit.mpt`. That is, if FIA was not obtained for the models, `select.mpt` only uses AIC and BIC. We strongly recommend using FIA for model selection (see e.g., Gruenwald, 2000).

The outputs follows the same principle for all information criteria. The lowest value is taken as the reference value and the differences to this value (i.e., the `delta`) are reported for all models (e.g., `delta.FIA`). If one additionally wants the original values, `output` needs to be set to `"full"`.

For AIC and BIC, AIC and BIC weights are reported as `wAIC` and `wBIC` (Wagenmakers & Farrell, 2004).

For multi-individual fit, `select.mpt` will additionally return how often each model provided the best fit (e.g., `FIA.best`). Values are rounded before determining which is the best fitting model. Note that there can be ties so that two models provide the best fit. Furthermore, if `output` is `"standard"`, only results for the summed information criteria are returned (indicated by the postfix `.sum`). To obtain model selection results for the aggregated data (indicated by postfix `.aggregated`), `output` needs to be set to `"full"`.

select.mpt will check if the data of the results returned from `fit.mpt` are equal. (If they are not equal model selection can not be done.)

Note that the values in the returned `data.frame` are rounded to the `round.digitth` decimal place.

Value

A `data.frame` containing the model selection values:
`model`: Name or number of model (names are either taken from `mpt.results` or obtained via `match.call`).
`n.parameters`: Number of parameters for each model.
`G.Squared`: G.Squared values of the model (from summed fits for multiple datasets).
`df`: df values of the model (from summed fits for multiple datasets).
`p.value`: p values of the model (from summed fits for multiple datasets).
`p.smaller.05`: How many of the individual data sets have p < .05 (for multiple datasets only).
For the information criteria (i.e., FIA, AIC, BIC) `X`, `delta.X`, `X.best`, `X`, `wX` represent: The difference from the reference model, how often each model provided the best fit (only for multi-individual fit), the absolute value, the weights (only AIC and BIC).
For multi-indivudal fit the postfix indicates whether the results refer to the summed information criteria from individual fit `.sum` or the information criteria from the aggregated data `.aggregated`.

Note

As of March 2015 BIC and FIA are calculated anew if the results are displayed for multiple data sets as BIC and FIA cannot directly be summed across participants due to the $log(n)$ terms in their formula (while AIC can be summed). Instead one first needs to sum the G^2 values, n, and the number of parameters, and only then can BIC and FIA be calculated for those summed values.

If any of the models is fitted with `fit.aggregated = FALSE` no aggregated results are presented.

Author(s)

Henrik Singmann

References

Gruenwald, P.D. (2000). Model selection based on minimum description length. *Journal of Mathematical Psychology*, 44, 133-152.

Wagenmakers, E.J. & Farrell, S. (2004). AIC model selection using Akaike weights. *Psychonomic Bulletin & Review*, 11, 192-196.

See Also

`fit.mpt` for obtaining the results needed here and an example using multi-individual fit and FIA.

Examples

```
# This example compares the three versions of the model in
# Riefer and Batchelder (1988, Figure 2)

data(rb.fig2.data)
model2 <- system.file("extdata", "rb.fig2.model", package = "MPTinR")
model2r.r.eq <- system.file("extdata", "rb.fig2.r.equal", package = "MPTinR")
model2r.c.eq <- system.file("extdata", "rb.fig2.c.equal", package = "MPTinR")

# The full (i.e., unconstrained) model
ref.model <- fit.mpt(rb.fig2.data, model2)
# All r equal
r.equal <- fit.mpt(rb.fig2.data, model2, model2r.r.eq)
# All c equal
c.equal <- fit.mpt(rb.fig2.data, model2, model2r.c.eq)

select.mpt(list(ref.model, r.equal, c.equal))



## Not run:

# Example from Broder & Schutz (2009)

data(d.broeder, package = "MPTinR")
m.2htm <- system.file("extdata", "5points.2htm.model", package = "MPTinR")
r.2htm <- system.file("extdata", "broeder.2htm.restr", package = "MPTinR")
r.1htm <- system.file("extdata", "broeder.1htm.restr", package = "MPTinR")

br.2htm.fia <- fit.mpt(d.broeder, m.2htm, fia = 50000, fit.aggregated = FALSE)
br.2htm.res.fia <- fit.mpt(d.broeder, m.2htm, r.2htm, fia = 50000, fit.aggregated = FALSE)
br.1htm.fia <- fit.mpt(d.broeder, m.2htm, r.1htm, fia = 50000, fit.aggregated = FALSE)

select.mpt(list(br.2htm.fia, br.2htm.res.fia, br.1htm.fia))
# This table shows that the n (number of trials) is too small to correctly compute
# FIA for the 1HT model (as the penalty for the 1HTM is larger than for the 2HTM,
# although the former is nested in the latter).
# This problem with FIA can only be overcome by collecting more trials per participant,
# but NOT by collecting more participants (as the penalties are simply summed).
```

```
# using the dataset argument we see the same
select.mpt(list(br.2htm.fia, br.2htm.res.fia, br.1htm.fia), dataset = 4, output = "full")

select.mpt(list(br.2htm.fia, br.2htm.res.fia, br.1htm.fia),dataset = 1:10)

## End(Not run)
```

Index