Package ‘Infusion’

May 3, 2023

Type Package

Title Inference Using Simulation

Description Implements functions for simulation-based inference. In particular, implements functions to perform likelihood inference from data summaries whose distributions are simulated. A first approach was described in Rousset et al. (2017 <doi:10.1111/1755-0998.12627>) but the package implements more advanced methods.

Encoding UTF-8

Version 2.1.0

Date 2023-05-03

Imports spaMM (>= 4.1.66), proxy, blackbox (>= 1.1.41), mvtnorm, methods, numDeriv, viridis, pbapply, ranger, foreach, matrixStats

Suggests testthat, Rmixmod, crayon, caret, xLLiM

Depends R (>= 3.3.0)

Maintainer François Rousset <francois.rousset@umontpellier.fr>

License CeCILL-2

ByteCompile true

URL https://www.R-project.org, https://gitlab.mbb.univ-montp2.fr/francois/Infusion

NeedsCompilation no

Author François Rousset [aut, cre, cph] (<https://orcid.org/0000-0003-4670-0371>)

Repository CRAN

Date/Publication 2023-05-03 11:00:03 UTC

R topics documented:

`add_reftable` *Create or augment a list of simulated distributions of summary statistics*

Description

`add_reftable` creates or augments a reference table of simulations, and formats the results appropriately for further use. The user does not have to think about this return format. Instead, s-he only has to think about the very simple return format of the function given as its `Simulate` argument. The primary role of his function is to wrap the call(s) of the function specified by `Simulate`. Depending on the arguments, parallel or serial computation is performed.

When parallelization is implied, it is performed by by default a "socket" cluster, available on all operating systems. Special care is then needed to ensure that all required packages are loaded in the called processes, and that all required variables and functions are passed therein: check the packages and env arguments. For socket clusters, foreach or pbapply is called depending whether the doSNOW package is attached (doSNOW allows more efficient load balancing than pbapply).

Alternatively, if the simulation function cannot be called directly by the R code, simulated samples can be added using the newsimuls argument. Finally, a generic data frame of simulated samples can be reformatted as a reference table by using only the reftable argument.

add_simulation is a wrapper for add_reftable, suitable when nRealizations>1. It is now distinctly documented: the distinct features of add_simulation were conceived for the first workflow implemented in Infusion but are somewhat obsolete now.

Usage

```
add_reftable(reftable=NULL, Simulate, parsTable=par.grid, par.grid=NULL,
                 nRealizations = 1L, newsimuls = NULL,
              verbose = interactive(), nb_cores = NULL, packages = NULL, env = NULL,
                 control.Simulate=NULL, cluster_args=list(), cl_seed=NULL, ...)
```

Arguments

reftable	Data frame: a reference table. Each row contains parameters value of a simulated realization of the data-generating process, and the simulated summary statistics. As parameters should be told apart from statistics by **Infusion** functions, information about parameter names should be attached to the reftable ***if*** it is not available otherwise. Thus if no parsTable is provided, the reftable should have an attribute "LOWER" (a named vectors giving lower bounds for the parameters which will vary in the analysis, as in the return value of the function).
Simulate	An *R* function, or the name (as a character string) of an *R* function used to generate summary statistics for samples form a data-generating process. When an external simulation program is called, Simulate must therefore be an R function wrapping the call to the external program. Two function APIs are handled: * If the function has a parsTable argument, it must return a data frame of summary statistics, each line of which contains the vector of summary statistics for one realization of the data-generating process. The parsTable argument of add_reftable will be passed to Simulate and lines of the output data frame must be ordered, as in the input parsTable as these two data frames will be bound together. * Otherwise, the Simulate function must have one argument for each element of the parameter vector (i.e. of each row of parsTable). It must return a vector of summary statistics with named vector member.
parsTable, par.grid	A data frame of which each line is the vector of parameters needed by Simulate for each simulation of the data-generating process. par.grid is an alias for parsTable; the latter argument may be preferred in order not to suggest that the parameter values should form a regular grid.
nRealizations	The number of simulated samples of summary statistics, for each parameter vector (each row of parsTable). If not 1, theold wrkflow is assumed and add_simulation is called.

`newsimuls`	If the function used to generate empirical distributions cannot be called by R, then `newsimuls` can be used to provide these distributions. See Details for the structure of this argument.
`nb_cores`	Number of cores for parallel simulation; `NULL` or integer value, acting as a shortcut for `cluster_args$spec`. This is effective only if the simulation function is called separately for each row of `parsTable`. Otherwise, if the simulation function is called once one the whole `parsTable`, parallelisation could be controlled only through that function's own arguments.
`cluster_args`	A list of arguments, passed to `makeCluster`. May contain a non-null `spec` element, in which case the distinct `nb_cores` argument and the global **Infusion** option `nb_cores` are ignored. A typical usage would thus be `control_args=list(spec=<number of 'children'>)`. Additional elements `outfile="log.txt"` may be useful to collect output from the nodes, and `type="FORK"` may be used to force a fork cluster on linux(-alikes) (otherwise a socket cluster is set up as this is the default effect of `parallel::makeCluster`). Do ***not*** use a structured list with an `add_reftable` element as is possible for `refine` (see Details of `refine` documentation).
`verbose`	Whether to print some information or not.
`...`	Additional arguments passed to `Simulate`, beyond the parameter vector. These arguments should be constant through all the simulation workflow.
`control.Simulate`	A list, used as an exclusive alternative to "..." to pass additional arguments to `Simulate`, beyond the parameter vector. The list must contain the same elements as would otherwise go in the "..." (if `control.Simulate` is left NULL, a default value is constructed from the ...).
`packages`	For parallel evaluation: Names of additional libraries to be loaded on the cores, necessary for `Simulate` evaluation.
`env`	For parallel evaluation: an environment containing additional objects to be exported on the cores, necessary for `Simulate` evaluation.
`cl_seed`	(all parallel contexts:) Integer, or NULL. If an integer, it is used to initialize `"L'Ecuyer-CMRG"` random-number generator. If `cl_seed` is `NULL`, the default generator is selected on each node, where its seed is not controlled. Providing the seed allows repeatable results for given parallelization settings, but may not allow identical results across different settings.

Details

The `newsimuls` argument should have the same structure as the return value of the function itself, except that `newsimuls` may include only a subset of the attributes returned by the function. It is thus a data frame; its required attributes are `LOWER` and `UPPER` which are named vectors giving bounds for the parameters which are variable in the whole analysis (note that the names identify these parameters in the case this information is not available otherwise from the arguments). The values in these vectors may be incorrect in the sense of failing to bound the parameters in the `newsimuls`, as the actual bounds are then corrected using parameter values in `newsimuls` and attributes from `reftable`.

Value

A data.frame (with additional attributes) is returned.

The value has the following attributes: `LOWER` and `UPPER` which are each a vector of per-parameter minima and maxima deduced from any `newsimuls` argument, and optionally any of the arguments `Simulate`, `control.Simulate`, `packages`, `env`, `parsTable` and `reftable` (all corresponding to input arguments when provided, except that the actual `Simulate` function is returned even if it was input as a name).

Examples

```
## see main documentation page for the package for other typical usage
```

`add_simulation` *Create or augment a list of simulated distributions of summary statistics*

Description

`add_simulation` is suitable for the primitive **Infusion** workflow; otherwise, it is cleaer to call `add_reftable` directly. `add_simulation` creates or augments a list of simulated distributions of summary statistics, and formats the results appropriately for further use. Alternatively, if the simulation function cannot be called directly by the R code, simulated distributions can be added using the `newsimuls` argument, using a simple format (see `onedistrib` in the Examples). Finally, a generic data frame of simulations can be reformatted as a reference table by using only the `simulations` argument.

Depending on the arguments, parallel or serial computation is performed. When parallelization is implied, by default a “socket” cluster, available on all operating systems. Special care is then needed to ensure that all required packages are loaded in the called processes, and that all required variables and functions are passed therein: check the `packages` and `env` arguments. For socket clusters, `foreach` or `pbapply` is called depending whether the `doSNOW` package is attached (`doSNOW` allows more efficient load balancing than `pbapply`).

Usage

```
add_simulation(simulations=NULL, Simulate, parsTable=par.grid, par.grid=NULL,
               nRealizations=Infusion.getOption("nRealizations"),
               newsimuls=NULL, verbose=interactive(), nb_cores=NULL,
               packages=NULL, env=NULL, control.Simulate=NULL,
               cluster_args=list(), cl_seed=NULL, ...)
```

Arguments

`simulations`	A list of matrices each representing a simulated distribution for given parameters in a format consistent with the return format of `add_simulation`.
`nRealizations`	The number of simulated samples of summary statistics, for each empirical distribution (each row of `par.grid`).

`Simulate`	An *R* function, or the name (as a character string) of an *R* function used to generate empirical distributions of summary statistics. When an external simulation program is called, `Simulate` must therefore be an R function wrapping the call to the external program. The `Simulate` function must have one argument for each element of the parameter vector (i.e. of each row of `par.grid`). It must return a vector of summary statistics with named vector members; **or** a single matrix of `nRealizations` simulations, in which case its rows and row names must represent the summary statistics, it should have `nRealizations` columns, and `nRealizations` should be named integer of the form "`c(as_one=.)`" (see Examples).
`parsTable, par.grid`	A data frame of which each line is the vector of parameters needed by `Simulate` for each simulation of the data-generating process. `par.grid` is an alias for `parsTable`; the latter argument may be preferred in order not to suggest that the parameter values should form a regular grid.
`newsimuls`	If the function used to generate empirical distributions cannot be called by R, then `newsimuls` can be used to provide these distributions. See Details for the structure of this argument.
`nb_cores`	Number of cores for parallel simulation; `NULL` or integer value, acting as a shortcut for `cluster_args$spec`. The effect is complicated: see Details.
`cluster_args`	A list of arguments, passed to `makeCluster`. May contain a non-null `spec` element, in which case the distinct `nb_cores` argument and the global **Infusion** option `nb_cores` are ignored. A typical usage would thus be `control_args=list(spec=<number of 'children'>)`. Additional elements `outfile="log.txt"` may be useful to collect output from the nodes, and `type="FORK"` may be used to force a fork cluster on linux(-alikes) (otherwise a socket cluster is set up as this is the default effect of `parallel::makeCluster`).
`verbose`	Whether to print some information or not.
`...`	Arguments passed to `add_reftable` (and possibly beyond, to the simulation function: see `nsim` argument of `myrnorm_tab()` in the Examples. These arguments should be constant through all the simulation workflow.
`control.Simulate`	A list, used as an exclusive alternative to "..." to pass additional arguments to `Simulate`, beyond the parameter vector. The list must contain the same elements as would go in the "...".
`packages`	For parallel evaluation: Names of additional libraries to be loaded on the cores, necessary for `Simulate` evaluation.
`env`	For parallel evaluation: an environment containing additional objects to be exported on the cores, necessary for `Simulate` evaluation.
`cl_seed`	Integer, or NULL. Providing the seed was conceived to allow repeatable results at least for given parallelization settings, if not identical results across different parallelization contexts. However, this functionality may have been been lost as the code was adapted for the up-to-date workflow using `add_reftable`.

Details

The newsimuls argument should have the same structure as the return value of the function itself, except that newsimuls may include only a subset of the attributes returned by the function. newsimuls should thus be list of matrices, each with a par attribute (see Examples). Rows of each matrix stand for simulation replicates and columns stand for the different summary statistics.

When nRealizations>1L, if nb_cores is unnamed or has name "replic" and if the simulation function does not return a single table for all replicates (thus, if nRealizations is **not** a named integer of the form "c(as_one=.)", parallelisation is over the different samples for each parameter value (and the seed of the random number generator is not controlled in a parallel context). For any other explicit name (e.g., nb_cores=c(foo=7)), or if nRealizations is a named integer of the form "c(as_one=.)", parallelisation is over the parameter values (the rows of par.grid). In all cases, the progress bar is over parameter values. See Details in Infusion.options for the subtle way these different cases are distinguished in the progress bar.

Using a FORK cluster with nRealizations>1 is warned as unreliable: in particular, anyone trying this combination should check whether other desired controls, such as random generator seed, or progress bar are effective.

Value

If nRealizations>1L, the return value is an object of class EDFlist, which is a list-with-attributes of matrices-with-attribute. Each matrix contains a simulated distribution of summary statistics for given parameters, and the "par" attribute is a 1-row data.frame of parameters. If Simulate is used, this must give all the parameters to be estimated; otherwise it must at least include all variable parameters in this **or later** simulations to be appended to the simulation list.

The value has the following attributes: LOWER and UPPER which are each a vector of per-parameter minima and maxima deduced from any newsimuls argument, and optionally any of the arguments Simulate, control.Simulate, packages, env, par.grid and simulations (all corresponding to input arguments when provided, except that the actual Simulate function is returned even if it was input as a name).

If nRealizations=1 add_reftable is called: see its distinct return value.

Examples

```
### Examples using init_grid and add_simulation, for primitive workflow
### Use init_reftable and add_reftable for the up-to-date workflow

# example of building a list of simulations from scratch:
myrnorm <- function(mu,s2,sample.size) {
  s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
  return(c(mean=mean(s),var=var(s)))
}
set.seed(123)
onedistrib <- t(replicate(100,myrnorm(1,1,10))) # toy example of simulated distribution
attr(onedistrib,"par") <- c(mu=1,sigma=1,sample.size=10) ## important!
simuls <- add_simulation(NULL, Simulate="myrnorm", nRealizations=500,
                         newsimuls=list("example"=onedistrib))

# standard use: smulation over a grid of parameter values
```

```
parsp <- init_grid(lower=c(mu=2.8,s2=0.2,sample.size=40),
                   upper=c(mu=5.2,s2=3,sample.size=40))
simuls <- add_simulation(NULL, Simulate="myrnorm", nRealizations=500,
                         par.grid = parsp[1:7,])

## Not run:  # example continued: parallel versions of the same
# Slow computations, notably because cluster setup is slow.

#    ... parallel over replicates, serial over par.grid rows
# => cl_seed has no effect and can be ignored
simuls <- add_simulation(NULL, Simulate="myrnorm", nRealizations=500,
                         par.grid = parsp[1:7,], nb_cores=7)
#
#    ... parallel over 'par.grid' rows => cl_seed is effective
simuls <- add_simulation(NULL, Simulate="myrnorm", nRealizations=500,
                         cl_seed=123, # for repeatable results
                         par.grid = parsp[1:7,], nb_cores=c(foo=7))

## End(Not run)

####### Example where a single 'Simulate' returns all replicates:

myrnorm_tab <- function(mu,s2,sample.size, nsim) {
  ## By default, Infusion.getOption('nRealizations') would fail on nodes!
  replicate(nsim,
            myrnorm(mu=mu,s2=s2,sample.size=sample.size))
}

parsp <- init_grid(lower=c(mu=2.8,s2=0.2,sample.size=40),
                   upper=c(mu=5.2,s2=3,sample.size=40))

# 'as_one' syntax for 'Simulate' function returning a simulation table:
simuls <- add_simulation(NULL, Simulate="myrnorm_tab",
              nRealizations=c(as_one=500),
              nsim=500, # myrnorm_tab() argument, part of the 'dots'
              par.grid=parsp)

## Not run:  # example continued: parallel versions of the same.
# Slow cluster setup again
simuls <- add_simulation(NULL,Simulate="myrnorm_tab",par.grid=parsp,
              nb_cores=7L,
              nRealizations=c(as_one=500),
              nsim=500, # myrnorm_tab() argument again
              cl_seed=123, # for repeatable results
              # need to export other variables used by *myrnorm_tab* to the nodes:
              env=list2env(list(myrnorm=myrnorm)))

## End(Not run)

## see main documentation page for the package for other typical usage
```

`check_raw_stats` *Check linear dependencies among raw summary statistics*

Description

A convenient wrapper function for `caret::findLinearCombos`, allowing to detect linear dependencies among the statistics, and optionally to remove variables that induce them.

Usage

```
check_raw_stats(x, statNames, remove = FALSE, verbose = interactive())
```

Arguments

`x`	data frame (particularly inheriting from class `"reftable"`, i.e. a reference table of simulations); or possibly a matrix with column names
`statNames`	Character vector: variables among which dependencies are sought. Must belong column names of `x`. For a `reftable`, this argument is optional and by default, all raw statistic are included. For other classes of input, this argument is required.
`remove`	Boolean: whether to return `x` with “offending” columns removed, or other information.
`verbose`	Boolean: whether to display some messages.

Value

Return type depends on the availability of the **caret** package, and on the `remove` argument, as follows. if `remove=TRUE`, an object of the same class as `x` is returned (with redundant columns removed). If `remove=FALSE`, either the **caret** package is available, in which case a list is returned with the same structure as the return value of `caret::findLinearCombos` but with column indices replaced by column names; or a message pointing that **caret** is not available is returned (and another is printed, only once per session).

`confint.SLik` *Compute confidence intervals by (profile) summary likelihood*

Description

This takes an `SLik` object (as produced by `MSL`) and deduces confidence bounds for each parameter, using a (profile, if relevant) likelihood ratio method.

Usage

```
## S3 method for class 'SLik'
confint(object, parm,
                        level=0.95, verbose=interactive(),
                        fixed=NULL,which=c(TRUE,TRUE),...)
```

Arguments

`object`	an `SLik` or `SLik_j` object
`parm`	The parameter which confidence bounds are to be computed
`level`	The desired coverage of the interval
`verbose`	Whether to print some information or not
`fixed`	When this is `NULL` the computed interval is a profile confidence interval over all parameters excluding `parm`. `fixed` allows one to set fixed values to some of these parameters.
`which`	A pair of booleans, controlling whether to compute respectively the lower and the upper CI bounds.
`...`	further arguments passed to or from other methods (currently not used).

Value

A list with sublists for each parameter, each sublist containing of three vectors: the bounds of the one-dimensional confidence interval; the “full” (only parameters variable in the `SLik` object are considered) parameter point for the lower bound, and the full parameter point for the upper bound

Examples

```
## see main documentation page for the package
```

`densv` *Saved computations of inferred log-likelihoods*

Description

These are saved results from toy examples used in other documentation page for the package. It gives estimates by simulation of log-likelihoods of the `(mu,s2)` parameters of a Gaussian distribution for a given sample of size 20 with mean 4.1416238 and (bias-corrected) variance 0.9460778. `densv` is based on the sample mean and sample variance as summary statistics, and `densb` on more contrived summary statistics.

Usage

```
data("densv")
data("densb")
```

Format

Data frames (with additional attributes) with observations on the following 5 variables.

`mu` a numeric vector; mean parameter of simulated Gaussian samples

`s2` a numeric vector; variance parameter of simulated Gaussian samples

`sample.size` a numeric vector; size of simulated Gaussian samples

`logL` a numeric vector; log probability density of a given statistic vector inferred from simulated values for the given parameters

`isValid` a boolean vector. See `infer_logLs` for its meaning.

Both data frames are return objects of a call to `infer_logLs`, and as such they includes attributes providing information about the parameter names and statistics names (not detailed here).

See Also

See step (3) of the workflow in the Example on the main `Infusion` documentation page, showing how `densv` was produced, and the Example in `project` showing how `densb` was produced.

`dMixmod` *Internal S4 classes.*

Description

The objects or methods referenced here are not to be called by the user, or are waiting for documentation to be written.

`dMixmod` is an S4 class describing some distributions that extend the multivariate gaussian mixture models (MGMM) by possibly involving discrete probability masses for some variables and gaussian mixtures for other variables conditional on such discrete events. In terms of the represented probability models, and of its slots, is effectively extends the `MixmodResults` class from the `Rmixmod` package. But it does not formally extends this class in terms of OOP programming. It should not be considered as part of the programming interface, and may be subject to backward-incompatible modifications without notice. In the current implementation it cannot represent general mixtures of discrete probabilities and MGMMs, and may yield correct results only for the degenerate case of pure MGMMs or when inference can be based on the conditional density of continuous variables conditional on the (joint-, if relevant) discrete event observed in the data.

Usage

```
# dMixmod: Don't try to use it! It's for programming only.
```

Value

A dMixmod object has the same slots as a MixmodResults object, plus additional ones: @freq is the frequency of the conditioning event for the gaussian mixture model. In the Infusion code, this event is defined jointly by the “observed” summary statistics and the reference simulation table: a probability mass for specific values **v** is identified from the simulated distribution of summary statistics in the reference table, and freq is an estimate of the probability mass if the summary statistics match **v**, or the converse probability if they do not match.

Note

Use str(attributes(.)) to see the slots of a dMixmod object if str(.) does not work.

Examples

```
# The dMixmod object can be used internally to handle repeated and boundary values
# of summary statistics. The user has to add an attribute to the observations,
# as explained in help("boundaries-attribute"):
Sobs <- c(mean=4.321, se=0.987) # hypothetical observation
attr(Sobs,"boundaries") <- c(someSummStat=-1)
```

---

example_raw *Workflow for primitive method, without projections*

Description

Example of the workflow with add_simulation), implementing the method described in the original publication (Rousset et al. 2017 <doi:10.1111/1755-0998.12627>).

Examples

```
## The following example illustrates the workflow.
## However, most steps run longer than accepted by the CRAN checks,
## So by default they will not run.
##
## (1) The user must provide the function for simulation of summary statistics
myrnorm <- function(mu,s2,sample.size) {
 s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
 return(c(mean=mean(s),var=var(s)))
} # simulate means and variances of normal samples of size 'sample.size'
#
## pseudo-sample:
set.seed(123)
Sobs <- myrnorm(mu=4,s2=1,sample.size=40) ## stands for the actual data to be analyzed
#
## (2) Generate, and simulate distributions for,
##        an irregular grid of parameter values, with some replicates
if (Infusion.getOption("example_maxtime")>40) {
  parsp <- init_grid(lower=c(mu=2.8,s2=0.2,sample.size=40),
                     upper=c(mu=5.2,s2=3,sample.size=40))
```

```
    simuls <- add_simulation(NULL,Simulate="myrnorm",par.grid=parsp)

    ## (3) infer logL(pars,stat.obs) for each simulated 'pars'
    # Relatively slow, hence saved as data 'densv'
    densv <- infer_logLs(simuls,stat.obs=Sobs)
  } else {
    data(densv)
    .Random.seed <- saved_seed
  }
  #
  ## (4) infer a log-likelihood surface and its maximum;
  ##       plot and extract various information.
  if (Infusion.getOption("example_maxtime")>11) {
   slik <- infer_surface(densv)
   slik <- MSL(slik) ## find the maximum of the log-likelihood surface
   plot(slik)
   profile(slik,c(mu=4)) ## profile summary logL for given parameter value
   confint(slik,"mu") ## compute confidence interval for given parameter
   plot1Dprof(slik,pars="s2",gridSteps=40) ## 1D profile
  }
  #
  ## (5) ## refine iteratively
  if (Infusion.getOption("example_maxtime")>39) {
   slik <- refine(slik)
  }
```

---

example_raw_proj *Workflow for primitive method, with projections*

Description

Example of the workflow with `add_simulation`), implementing the method described in the original publication (Rousset et al. 2017 <doi:10.1111/1755-0998.12627>), modified to use projectors.

Examples

```
if (Infusion.getOption("example_maxtime")>170) {
## Normal(mu,sd) model, with inefficient raw summary statistics:
## To illustrate that case we transform normal random deviates rnorm(,mu,sd)
## so that the mean of transformed sample is not sufficient for mu,
## and the variance of transformed sample is not sufficient for sd.
blurred <- function(mu,s2,sample.size) {
  s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
  s <- exp(s/4)
  return(c(mean=mean(s),var=var(s)))
}

set.seed(123)
dSobs <- blurred(mu=4,s2=1,sample.size=20) ## stands for the actual data to be analyzed
## Sampling design as in canonical example
parsp <- init_grid(lower=c(mu=2.8,s2=0.4,sample.size=20),
                      upper=c(mu=5.2,s2=2.4,sample.size=20))
# simulate distributions
dsimuls <- add_simulation(,Simulate="blurred", par.grid=parsp)

## Use projection to construct better summary statistics for each each parameter
mufit <- project("mu",stats=c("mean","var"),data=dsimuls)
s2fit <- project("s2",stats=c("mean","var"),data=dsimuls)

## additional plots for some projection method
if (inherits(mufit,"HLfit")) mapMM(mufit,map.asp=1,
  plot.title=title(main="prediction of normal mean",xlab="exp mean",ylab="exp var"))
if (inherits(s2fit,"HLfit")) mapMM(s2fit,map.asp=1,
  plot.title=title(main="prediction of normal var",xlab="exp mean",ylab="exp var"))

## apply projections on simulated statistics
corrSobs <- project(dSobs,projectors=list("MEAN"=mufit,"VAR"=s2fit))
corrSimuls <- project(dsimuls,projectors=list("MEAN"=mufit,"VAR"=s2fit))

## Analyze 'projected' data as any data (cf canonical example)
densb <- infer_logLs(corrSimuls,stat.obs=corrSobs)
} else data(densb)
#########
if (Infusion.getOption("example_maxtime")>10) {
slik <- infer_surface(densb) ## infer a log-likelihood surface
slik <- MSL(slik) ## find the maximum of the log-likelihood surface
}
if (Infusion.getOption("example_maxtime")>500) {
slik <- refine(slik,10, update_projectors=TRUE) ## refine iteratively
}
```

example_reftable *Workflow for method with reference table*

Description

Examples of workflow with a reference table produced by `add_reftable`, possibly faster in many applications than the originally described method.

Examples

```
if (Infusion.getOption("example_maxtime")>46) {

  ## Normal(mu,sd) model, with inefficient raw summary statistics:
  ## To illustrate that case we transform normal random deviates rnorm(,mu,sd)
  ## so that the mean of transformed sample is not sufficient for mu,
  ## and the variance of transformed sample is not sufficient for sd.
  blurred <- function(mu,s2,sample.size) {
    s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
    s <- exp(s/4)
    return(c(mean=mean(s),var=var(s)))
  }

  ## pseudo-sample which stands for the actual data to be analyzed:
  set.seed(123)
  dSobs <- blurred(mu=4,s2=1,sample.size=40)

  ## Construct reference table:
  parsp_j <- data.frame(mu=runif(600L,min=2.8,max=5.2),
                        s2=runif(600L,min=0.4,max=2.4),sample.size=40)
  dsimuls <- add_reftable(,Simulate="blurred",par.grid=parsp_j,verbose=FALSE)

  #- When no 'Simulate' function is provided,
  #- but only a data.frame 'toydf' of simulations,
  #- a formal reference table can be produced by
  # dsimuls <- structure(toydf, LOWER=c(mu=2,s2=0,sample.size=40))
  # dsimuls <- add_reftable(dsimuls)
  #- where the 'LOWER' attribute tells
  #- the parameters apart from the summary statistics.

  ## Construct projections
  mufit <- project("mu",stats=c("mean","var"),data=dsimuls,verbose=FALSE)
  s2fit <- project("s2",stats=c("mean","var"),data=dsimuls,verbose=FALSE)
  dprojectors <- list(MEAN=mufit,VAR=s2fit)

  ## Apply projections on simulated statistics and 'data':
  dprojSimuls <- project(dsimuls,projectors=dprojectors,verbose=FALSE)
  dprojSobs <- project(dSobs,projectors=dprojectors)

  ## Summary-likelihood inference:
  # Infer log-likelihood surface
  slik_j <- infer_SLik_joint(dprojSimuls,stat.obs=dprojSobs,verbose=TRUE)
  # Find maximum, confidence intervals...
  slik_j <- MSL(slik_j)

  # Convenience function for plotting projections...
  plot_proj(slik_j, parm="mu", proj="MEAN")

  # ... and for computing likelihoods for new parameters and/or data:
  summLik(slik_j, parm=slik_j$MSL$MSLE+0.1)

  ## refine estimates iteratively
  slik_j <- refine(slik_j,maxit=5, update_projectors=TRUE)

  if (Infusion.getOption("example_maxtime")>99) { # Post-fit procedures,
    #                                   all with distinct documentation:

    plot(slik_j)
    profile(slik_j,c(mu=4)) ## profile summary logL for given parameter value
    confint(slik_j,"mu") ## compute 1D confidence interval for given parameter
    plot1Dprof(slik_j,pars="s2",gridSteps=40) ## 1D profile
    summary(slik_j) # or print()
```

```
    logLik(slik_j)

    SLRT(slik_j, h0=slik_j$MSL$MSLE+0.1, nsim = 100L) # LRT
    SLRT(slik_j, h0=slik_j$MSL$MSLE[1]+0.1, nsim = 100L) # profile LRT

    goftest(slik_j) # goodness of fit test

   # Low-level predict() method (rarely directly used, otherwise see its documentation!)
    predict(slik_j, newdata = slik_j$MSL$MSLE)        # the 'data' are here parameters!


  }
}
```

---

extractors *Summary, print and logLik methods for Infusion results.*

Description

summary prints information about the fit. print is an alias for summary. logLik extracts the log-likelihood (exact or approximated).

Usage

```
## S3 method for class 'SLik'
summary(object, ...)
## S3 method for class 'SLik'
print(x, ...)
## S3 method for class 'SLik'
logLik(object, ...)
# and identical usage for 'SLik_j' objects
```

Arguments

object, x	An object of class SLik or SLik_j;
...	further arguments passed to or from other methods (currently without any specific effect).

Value

logLik returns the inferred likelihood maximum, with attribute RMSE giving its root means square error of estimation. summary and summary return the object invisibly. They print details of the fits in a convenient form.

Note

See workflow example in example_reftable.

See Also

See `get_from` for a more general interface for extracting elements from Infusion results, and `summLik` for using a fit object to evaluate the likelihood function for distinct parameter values and even distinct data.

Examples

```
# See Note
```

`focal_refine` *Refine summary likelihood profile in focal parameter values*

Description

This function refines an `SLik_j` object in a focused way defined by focal parameter values. It is paticularly useful to check a suspect pattern in a likelihood profile. If there is a suspect dip or peak at value <somepar>=<somevalue>, `focal_refine(`<SLik_j object>`, focal=`c(<somepar>=<somevalue>)`,` `size=`<size>) will define <size> parameter points near `c(`<somepar>=<somevalue>) and (subject to these points being in the parameter bounds of the object) simulate new samples for these parameter points and refine the object using these new simulations.

Usage

```
focal_refine(object, focal, size, plotprof = TRUE, ...)
```

Arguments

`object`	An object of class `SLik_j`.
`focal`	Parameter value(s) (as a vector of named values)
`size`	Target number of points to add to the reference table
`plotprof`	Whether to replot a likelihood profile (1D or 2D depending on the dimension of `focal`).
`...`	Further arguments passed to `profile.SLik_j` (but not including argument `return.optim`) or `refine`.

Value

The updated `object`

Examples

```
## Not run:
# Using the slik_j object from the toy example in help("example_reftable"):

plot1Dprof(slik_j,"s2")
slik_fix <- focal_refine(slik_j,focal=c(s2=2), size=100)
plot1Dprof(slik_fix,"s2")

# In that case the effect is not spectacular because
# there is no major problem in the starting profile.

## End(Not run)
```

`get_from` *Backward-compatible extractor from summary-likelihood objects*

Description

A generic function, whose default method works for list, and with specific methods for objects inheriting from classes `SLik_j` and `SLik`.

Usage

```
get_from(object, which, ...)

## S3 methods with additional argument(s)
## S3 method for class 'SLik'
get_from(object, which, raw=FALSE, force=FALSE, ...)
## S3 method for class 'SLik_j'
get_from(object, which, raw=FALSE, force=FALSE, ...)
```

Arguments

`object`	Any object with a list structure.
`which`	Character: names of element to be extracted.
`raw`	Boolean: if TRUE, `object[[which]]` is returned, without any particular check of its value. By default, `raw` is FALSE and various operations may be performed on the extracted value (see “example” below), including optional recomputation if `force` is TRUE.
`force`	Boolean: if TRUE, the extracted element may be computed if it appears to be missing from the `object`. This is notably so for `which="RMSEs"` or `which="par_RMSEs"`; in these cases, the results of the computation are further saved in the original object.
`...`	further arguments passed to or from other methods (currently not used).

Value

Will depend on `which`, but aims to retain a convenient format backward compatible with version 1.4.0.

See Also

`logLik`.

Examples

```
# # Observed summary statistics
# #  (projected, with raw ones as attribute, if relevant)
#   get_from(slik, "obs")
#
# # On any summary-likelihood object 'slik':
#   get_from(slik, which="par_RMSEs") # matrix
# # despite <object>$par_RMSEs being an environment if
# #  'slik' was created by version > 1.4.0, as then shown by
#   get_from(slik, which="par_RMSEs", raw=TRUE)
#
# # Further, if
#   get_from(slik, which="par_RMSEs")
# # returns NULL because the element is absent from the object,
# # then one can force its computation by
#   get_from(slik, which="par_RMSEs", force=TRUE)
# # The result are saved in the 'slik' object, so running again
#   get_from(slik, which="par_RMSEs")
# # will no longer return NULL.
```

`get_LRboot` *Summary likelihood ratio tests*

Description

`get_LRboot` provides a fast approximation to bootstrap distribution of likelihood ratio statistic. The bootstrap distribution of the likelihood ratio (LR) statistic may be used to correct the tests based on its asymptotic chi-square distribution. However, the standard bootstrap involves resimulating the data-generating process, given the ML estimates on the original data. This function implements a fast approximation avoiding such simulation, instead drawing from the inferred distribution of (projected, if relevant) summary statistics, again given the maximum (summary-)likelihood estimates.

`SLRT` computes likelihood ratio tests based on the summary-likelihood surface and optionally on `get_LRboot` results. Several correction of the basic likelihood ratio test may be reported, some more speculative than others.

Usage

```
SLRT(object, h0, nsim=0L, BGP=NULL, ...)
get_LRboot(object, h0_pars = NULL, nsim = 100L, reset = TRUE, BGP=object$MSL$MSLE, ...)
```

Arguments

`object`	an `SLik_j` object.
`h0`	Numeric named vector of tested parameter values.
`h0_pars`	either `NULL` (the default), to approximate the distribution of the LR statistic for the full vector of estimated parameters; or a vector of names of a subset of this vector, to approximate the distribution of the profile LR statistic for this subset.
`nsim`	Integer: number of bootstrap replicates. Values lower than the default are not recommended. Note that this will be ignored if the distribution has previously been simulated and `reset=FALSE`.
`reset`	Boolean: Whether to use any previously computed distribution (see Details) or not.
`BGP`	Named numeric vector of "Bootstrap-Generating Parameters". Ideally the distribution of the LR test statistic would be pivotal and thus the parameter values under which this distribution is simulated would not matter. In practice, simulating by default its distribution under the "best" available information (the MSLE for `get_LRboot`, or the specifically tested hypothesis defined by the `h0` argument of `SLRT`) may be more accurate than under alternative parametric values. For `h0` being an incomplete parameter vector and `BGP` is NULL (the default), `SLRT` will simulate under a completed parameter vector using estimates of other parameters maximizing the likelihood profile for `h0`.
`...`	For `SLRT`: further arguments passed to `get_LRboot`. For `get_LRboot`: further arguments controlling parallelization, including `nb_cores`. However, parallelization may be best ignored in most cases (see Details).

Details

The result of calling `get_LRboot` (either directly or through `SLRT`) with given `h0_pars` is stored in the `object` (until the next `refine`), and this saved result is returned by a next call to `get_LRboot` with the same `h0_pars` if `reset=FALSE`. The default is however to recompute the distribution (`reset=TRUE`).

Parallelization is possible but maybe not useful because computations for each bootstrap replicate are fast relative to parallelization overhead. It will be called when the ... arguments include an `nb_cores>1`. The ... may include further arguments passed to `dopar`, but among the `dopar` arguments, `iseed` will be ignored, and `fit_env` should not be used.

A raw bootstrap p-value can be computed from the simulated distribution as `(1+sum(t >= t0))/(N+1)` where `t0` is the original likelihood ratio, `t` the vector of bootstrap replicates and `N` its length. See Davison & Hinkley (1997, p. 141) for discussion of the adjustments in this formula. However, a sometimes more economical use of the bootstrap is to provide a Bartlett correction for the likelihood ratio test in small samples. According to this correction, the mean value m of the likelihood ratio statistic under the null hypothesis is computed (here estimated by simulation) and the original LR statistic is multiplied by n/m where n is the number of degrees of freedom of the test. Unfortunately, the underlying assumption that the corrected LR statistic follows the chi-square distribution does not always work well.

Value

get_LRboot returns a numeric vector representing the simulated distribution of the LR statistic, i.e. **twice** the log-likelihood difference, as directly used in pchisq() to get the p-value.

SLRT returns a list with the following element(s), each being a one-row data frame:

basicLRT	A data frame including values of the likelihood ratio chi2 statistic, its degrees of freedom, and the p-value;

and, if a bootstrap was performed:

BartBootLRT	A data frame including values of the Bartlett-corrected likelihood ratio chi2 statistic, its degrees of freedom, and its p-value;
rawBootLRT	A data frame including values of the likelihood ratio chi2 statistic, its degrees of freedom, and the raw bootstrap p-value;
safeBartBootLRT	equal to rawBootLRT if the mean bootstrap value of the LR statistic is lower than the number of degrees of freedom, and to BartBootLRT otherwise.

References

Bartlett, M. S. (1937) Properties of sufficiency and statistical tests. Proceedings of the Royal Society (London) A 160: 268-282.

Davison A.C., Hinkley D.V. (1997) Bootstrap methods and their applications. Cambridge Univ. Press, Cambridge, UK.

Examples

```
## See help("example_reftable") for SLRT() examples;
## continuing from there, after refine() steps for good results:
# set.seed(123);mean(get_LRboot(slik_j, nsim=500, reset=TRUE)) # close to df=2
# mean(get_LRboot(slik_j, h0_pars = "s2", nsim=500, reset=TRUE)) # close to df=1

## Not run:
### Simulation study of performance of the corrected LRTs:

## Same toy example as in help("example_reftable"):
blurred <- function(mu,s2,sample.size) {
    s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
    s <- exp(s/4)
    return(c(mean=mean(s),var=var(s)))
  }

## First build a largish reference table and projections to be used in all replicates
# Only the 600 first rows will be used as initial reference table for each "data"
#
set.seed(123)
#
parsp_j <- data.frame(mu=runif(6000L,min=2.8,max=5.2),
                      s2=runif(6000L,min=0.4,max=2.4),sample.size=40)
```

```
dsimuls <- add_reftable(,Simulate="blurred",par.grid=parsp_j,verbose=FALSE)
#
mufit <- project("mu",stats=c("mean","var"),data=dsimuls,verbose=TRUE)
s2fit <- project("s2",stats=c("mean","var"),data=dsimuls,verbose=TRUE)
dprojectors <- list(MEAN=mufit,VAR=s2fit)
dprojSimuls <- project(dsimuls,projectors=dprojectors,verbose=FALSE)

## Function for single-data analysis:
#
foo <- function(y, refine_maxit=0L, verbose=FALSE) {
  dSobs <- blurred(mu=4,s2=1,sample.size=40)
  ## ----Inference workflow-----------------------------------------------
  dprojSobs <- project(dSobs,projectors=dprojectors)
  dslik <- infer_SLik_joint(dprojSimuls[1:600,],stat.obs=dprojSobs,verbose=FALSE)
  dslik <- MSL(dslik, verbose=verbose, eval_RMSEs=FALSE)
  if (refine_maxit) dslik <- refine(dslik, maxit=refine_maxit)
  ## ---- LRT-----------------------------------------------
  lrt <- SLRT(dslik, h0=c(s2=1), nsim=200)
  c(basic=lrt$basicLRT$p_value,raw=lrt$rawBootLRT$p_value,
    bart=lrt$BartBootLRT$p_value,safe=lrt$safeBartBootLRT$p_value)
}

## Simulations using convenient parallelization interface:
#
# library(doSNOW) # optional
#
bootreps <- spaMM::dopar(matrix(1,ncol=200,nrow=1),            # 200 replicates of foo()
 fn=foo, fit_env=list(blurred=blurred, dprojectors=dprojectors, dprojSimuls=dprojSimuls),
  control=list(.errorhandling = "pass", .packages = "Infusion"),
  refine_maxit=5L,
  nb_cores=parallel::detectCores()-1L, iseed=123)
#
plot(ecdf(bootreps["basic",]))
abline(0,1)
plot(ecdf(bootreps["bart",]), add=TRUE, col="blue")
plot(ecdf(bootreps["safe",]), add=TRUE, col="red")
plot(ecdf(bootreps["raw",]), add=TRUE, col="green")
#
# Note that refine() iterations are important for good performance.
# Without them, even a larger reftable of 60000 lines
# may exhibit poor results for some of the CI types.

## End(Not run)
```

`get_nbCluster_range` *Control of number of components in Gaussian mixture modelling*

Description

These functions implement the default values for the number of components tried in Gaussian mixture modelling (matching the `nbCluster` argument of `Rmixmod::mixmodCluster()`). `get_nbCluster_range` allows the user to reproduce the internal rules used by **Infusion** to determine this argument. `seq_nbCluster` is a wrapper to the function defined by the `seq_nbCluster` global option of the package. Its default result is a sequence of integers determined by the number of rows of the data (see `Infusion.options`). `get_nbCluster_range()` further checks the feasibility of the values generated by `seq_nbCluster())`, using additional criteria involving the number of columns of the data to determine the maximum feasible number of clusters. This maximum is controlled by the function defined by the `maxnbCluster` global option of the package.

`refine_nbCluster` controls the default number of clusters of `refine`: it gets the range from `seq_nbCluster` and keeps only the maximum value of this range if this maximum is higher than the `onlymax` argument.

Adventurous users can change the rules used by **Infusion** by changing the global options `seq_nbCluster` and `maxnbCluster` (while conforming to the interfaces of these functions). Less ambitiously, they can for example use the maximum value of the result of `get_nbCluster_range()` as a single reasonable value for the `nbCluster` argument of `infer_SLik_joint`.

Usage

```
seq_nbCluster(nr)
refine_nbCluster(nr, onlymax=7)
get_nbCluster_range(projdata, nr = nrow(projdata), nc = ncol(projdata),
                    nbCluster = seq_nbCluster(nr))
```

Arguments

`projdata`	data frame: the data to be clustered, which typically include parameters and **projected** summary statistics;
`nr`	integer: number of rows of the data to be clustered;
`onlymax`	integer: see Description;
`nc`	integer: number of columns of the data to be clustered, typically **twice** the number of estimated parameters;
`nbCluster`	integer or vector of integers: candidate values, which feasability is checked by the function.

Value

An integer vector

Examples

```
# Determination of number of clusters when attempting to estimate
#   20 parameters from a reference table with 30000 rows:
seq_nbCluster(nr=30000L)
get_nbCluster_range(nr=30000L, nc=40L) # nc = *twice* the number of parameters
```

goftest — *Assessing goodness of fit of inference using simulation*

Description

A goodness-of-fit test is performed in the case projected statistics have been used for inference. Otherwise some plots of limited interest are produced.

Usage

```
goftest(object, nsim = 99L, method = "", stats=NULL, plot. = TRUE, nb_cores = NULL,
        Simulate = attr(object$logLs, "Simulate"),
        packages = attr(object$logLs, "packages"),
        env = attr(object$logLs, "env"), verbose = interactive(),
        cl_seed=.update_seed(object), get_gof_stats=.get_gof_stats)
```

Arguments

`object`	an `SLik` or `SLik_j` object.
`nsim`	Number of draws of summary statistics.
`method`	For development purposes, not documented.
`stats`	Character vector, or NULL: the set of summary statistics to be used to construct the test. If NULL, the union, across all projections, of the raw summary statistics used for projections is potentially used for goodness of fit; however, if this set is too large for gaussian mixture modelling, a subset of variable may be selected. How they are selected is not yet fully settled (see Details).
`plot.`	Control diagnostic plots. `plot.` can be of logical, character or numeric type. If `plot.` is `FALSE`, no plot is produced. If `plot.` is `TRUE` (the default), a data frame of up to 8 goodness-of-fit statistics (the statistics denoted *u* in Details) is plotted. If more than eight raw summary statistics (denoted *s* in Details) were used, then only the first eight *u* are retained (see Details for the ordering of the *u*s here). If `plot.` is a **numeric vector**, then *u*`[plot.]` are retained (possibly more than 8 statistics, as in the next case). If `plot.` is a **character vector**, then it is used to match the names of the *u* statistics (not of *s*) to be retained in the plot; the names of *u* are built from names of *s* by wrapping the latter within `"Res(".")"` (see axes labels of default plots for examples of valid names).
`nb_cores, Simulate, packages, env, verbose`	See same-named `add_simulation` arguments.
`cl_seed`	NULL or integer (see `refine` for Details).
`get_gof_stats`	function for selecting raw statistics (see Details).

Details

Testing goodness-of-fit: The test is somewhat heuristic but appears to give reasonable results (the Example shows how this can be verified). It assumes that all summary statistics are reduced to projections predicting all model parameters. It is then conceived as if any projection p predicting a parameter were a sufficient statistic for this parameter, given the information contained in the summary statistics $\mathbf{s}$ (this is certainly the ideal objective of machine-learning regression methods). Then a statistic u independent (under the fitted model) from all projections should be a suitable statistic for testing goodness of fit: if the model is correctly specified, the quantile of observed u, in the distribution of u under the fitted model, should be uniformly distributed over repeated sampling under the data-generating process. The procedure constructs statistics uncorrelated to all $\mathbf{p}$ (over repeated sampling under the fitted model) and proceeds as if they were independent from p (rather than simply uncorrelated). A number (depending on the size of the reference table) of statistics u uncorrelated to p are then defined. Each such statistic is obtained as the residual of the regression of a given raw summary statistic to all projections, where the regression input is a simulation table of `nsim` replicates of $\mathbf{s}$ under the fitted model, and of their projections $\mathbf{p}$ (using the "projectors" constructed from the full reference table). The latter regression involves one more, small-`nsim`, approximation (as it is the sample correlation that is zeroed) but using the residuals is crucially better than using the original summary statistics (as some ABC software may do). An additional feature of the procedure is to construct a single test statistic t from joint residuals $\mathbf{u}$, by estimating their joint distribution (using Gaussian mixture modelling) and letting t be the density of $\mathbf{u}$ in this distribution.

Selection of raw summary statistics: See the code of the `Infusion:::.get_gof_stats` function for the method used. It requires that `ranger` has been used to produce the projectors, and that the latter include variable importance statistics (by default, **Infusion** calls `ranger` with argument `importance="permutation"`). `.get_gof_stats` then selects the raw summary statistics with *least* importance over projections (this may not be optimal, and in particular appears redundant with the procedure described below to construct goodness-of-fit statistics from raw summary statistics; so this might change in a later version), and returns a vector of names of raw statistics, sorted by increasing least-importance. The number of summary statistics can be controlled by the global package option `gof_nstats_fn`, a function with arguments `nr` and `nstats` for, respectively, the number of simulations of the processus (as controlled by `goftest(.,nsim)`) and the total number of raw summary statistics used in the projections.

The **diagnostic plot** will show a data frame of residuals u of the summary statistics identified as the first elements of the vector returned by `Infusion:::.get_gof_stats`, i.e. again a set of raw statistics with least-importance over projectors.

Value

A list with currently a single element

`pval`	The p-value of the test (NULL if the test is not feasible).

Examples

```
### See end of example("example_reftable") for minimal example.

## Not run:
### Performance of GoF test over replicate draws from data-generating process
```

```
# First, run
example("example_reftable")
# (at least up to the final 'slik_j' object), then

# as a shortcut, the same projections will be used in all replicates:
dprojectors <- slik_j$projectors

set.seed(123)
gof_draws <- replicate(200, {
  cat(" ")
  dSobs <- blurred(mu=4,s2=1,sample.size=40)
  ## ----Inference workflow-----------------------------------------------
  dprojSobs <- project(dSobs,projectors=dprojectors)
  dslik <- infer_SLik_joint(dprojSimuls,stat.obs=dprojSobs,verbose=FALSE)
  dslik <- MSL(dslik, verbose=FALSE, eval_RMSEs=FALSE)
  ## ----GoF test-----------------------------------------------
  gof <- goftest(dslik,nb_cores = 1L, plot.=FALSE,verbose=FALSE)
  cat(unlist(gof))
  gof
})
# ~ uniform distribution under correctly-specified model:
plot(ecdf(unlist(gof_draws)))

## End(Not run)
```

handling_NAs — *Discrete probability masses and NA/NaN/Inf in distributions of summary statistics.*

Description

This explains the use of the boundaries attribute of observed statistics to handle (1) values of the summary statistics that can occur with some probability mass; (2) special values (NA/NaN/Inf) in distributions of summary statistics. This further explains why Infusion handles special values by removing affected distributions unless the boundaries attribute is used.

Details

Special values may be encountered in an analysis. For example, trying to estimate a regression coefficient when the predictor variable is constant may return a NaN. Since functions such as refine automatically add simulated distributions, this problem must be automatically handled by the user's simulation function or by the package functions, rather than by user's tinkering with the Infusion procedures.

The user must consider what s-he would do if actual data also included NA/NaN/Inf values. If such data would not be subject to a statistical analysis, then the simulation procedure must reflect that, otherwise the analysis will be biased. The processing of reference tables by Infusion functions applies na.omit() on the tables so any line containing NA's will be removed. The drawbacks are that the number of informative simulations is reduced and that inference will be difficult if the data-generating parameters were indeed prone to induce data that would not be subject to statistical analysis. Thus, it may be necessary to simulate alternative data until no special values are obtained and the target size of the simulated distribution is reached. One solution is for the user to write a simulation function that calls itself recursively until a valid summary statistic is produced. Care is then needed to avoid infinite recursion (which might well indicate unlikely parameter values).

Alternatively, if one considers that special values are informative about parameters (in the above example of a regression coefficient, if a constant predictor variable says something about the parameters), then NA/NaN/Inf must be replaced by a (fixed) dummy numerical value which is flagged to be distinctly handled, using the `boundaries` attribute of the observed summary statistics. The simulation function should return statistic `foo=-1` (say) instead of `foo=NaN`, and one should then set `attr(<observed>,"boundaries") <- c(foo=-1)`.

The boundary attribute is also useful to handle all values of the summary statistics that can occur with some probability mass. For example if the estimate `est_p` of a probability takes values 0 or 1 with positive probability, one should set `attr(<observed>,"boundaries") <- c(p_est=0,p_est=1)`.

infer_logLs — *Infer log Likelihoods using simulated distributions of summary statistics*

Description

For each simulated distribution of summary statistics, `infer_logLs` infers a probability density function, and the density of the observed values of the summary statistics is deduced. By default, inference of each density is performed by `infer_logL_by_Rmixmod`, which fits a distribution of summary statistics using procedures from the `Rmixmod` package.

Usage

```
infer_logLs(object, stat.obs,
            logLname = Infusion.getOption("logLname"),
            verbose = list(most=interactive(),
                           final=FALSE),
            method = Infusion.getOption("mixturing"),
            nb_cores = NULL, packages = NULL, cluster_args,
            ...)
infer_tailp(object, refDensity, stat.obs,
                tailNames=Infusion.getOption("tailNames"),
                verbose=interactive(), method=NULL, cluster_args, ...)
infer_logL_by_GLMM(EDF,stat.obs,logLname,verbose)
infer_logL_by_Rmixmod(EDF,stat.obs,logLname,verbose)
infer_logL_by_mclust(EDF,stat.obs,logLname,verbose)
infer_logL_by_Hlscv.diag(EDF,stat.obs,logLname,verbose)
```

Arguments

`object`	A list of simulated distributions (the return object of `add_simulation`)
`EDF`	An empirical distribution, with a required `par` attribute (an element of the `object` list).
`stat.obs`	Named numeric vector of observed values of summary statistics.
`logLname`	The name to be given to the log Likelihood in the return object, or the root of the latter name in case of conflict with other names in this object.
`tailNames`	Names of “positives” and “negatives” in the binomial response for the inference of tail probabilities.
`refDensity`	An object representing a reference density (such as an `HLfit` fit object or other objects with a similar `predict` method) which, together with the density inferred from each empirical density, defines a likelihood ratio used to define a rejection region.
`verbose`	A list as shown by the default, or simply a vector of booleans, indicating respectively whether to display (1) some information about progress; (2) a final summary of the results after all elements of `simuls` have been processed. If a count of ’outlier’(s) is reported, this typically means that `stat.obs` is not within the envelope of a simulated distribution (or whatever other meaning the user attaches to an `FALSE` `isValid` code: see Details)
`method`	A function for density estimation. See Description for the default behaviour and Details for the constraints on input and output of the function.
`nb_cores`	Number of cores for parallel computation. The default is `spaMM.getOption("nb_cores")`, and 1 if the latter is NULL. `nb_cores=1` which prevents the use of parallelisation procedures.
`cluster_args`	A list of arguments, passed to `makeCluster`. May contain a non-null `spec` element, in which case the distinct `nb_cores` argument is ignored.
`packages`	For parallel evaluation: Names of additional libraries to be loaded on the cores, necessary for evaluation of a user-defined ’method’.
`...`	further arguments passed to or from other methods (currently not used).

Details

By default, density estimation is based on `Rmixmod` methods. Other available methods are not routinely used and not all of `Infusion` features may work with them. The function `Rmixmod::mixmodCluster` is called, with arguments `nbCluster=seq_nbCluster(nr=nrow(data))` and `mixmodGaussianModel=Infusion.getOptio`
If `Infusion.getOption("seq_nbCluster")` specifies a sequence of values, then several clusterings are computed and AIC is used to select among them.

`infer_logL_by_GLMM`, `infer_logL_by_Rmixmod`, `infer_logL_by_mclust`, and `infer_logL_by_Hlscv.diag` are examples of the method that may be provided for density estimation. Other `methods` may be provided with the same arguments. Their return value must include the element `logL`, an estimate of the log-density of `stat.obs`, and the element `isValid` with values `FALSE/TRUE` (or 0/1). The standard format for the return value is `unlist(c(attr(EDF,"par"),logL,isValid=isValid))`.

`isValid` is primarily intended to indicate whether the log likelihood of `stat.obs` inferred by a given density estimation method was suitable input for inference of the likelihood surface. `isValid` has two effects: to distinguish points for which isValid is FALSE in the plot produced by `plot.SLik`; and more critically, to control the sampling of new parameter points within `refine` so that points for which isValid is FALSE are less likely to be sampled.

Invalid values may for example indicate a likelihood estimated as zero (since log(0) is not suitable input), or (for density estimation methods which may infer erroneously large values when extrapolating), whether `stat.obs` is within the convex hull of the EDF. In user-defined `methods`, invalid inferred logL should be replaced by some alternative low estimate, as all methods included in the package do.

The source code of `infer_logL_by_Hlscv.diag` illustrates how to test whether `stat.obs` is within the convex hull of the EDF, using functions `resetCHull` and `isPointInCHull` (exported from the `blackbox` package).

`infer_logL_by_Rmixmod` calls `Rmixmod::mixmodCluster` `infer_logL_by_mclust` calls `mclust::densityMclust`, `infer_logL_by_Hlscv.diag` calls `ks::kde`, and `infer_logL_by_GLMM` fits a binned distribution of summary statistics using a Poisson GLMM with autocorrelated random effects, where the binning is based on a tesselation of a volume containing the whole simulated distribution. Limited experiments so far suggest that the mixture models methods are fast and appropriate (`Rmixmod`, being a bit faster, is the default method); that the kernel smoothing method is more erratic and moreover requires additional input from the user, hence is not really applicable, for distributions in dimension d= 4 or above; and that the GLMM method is a very good density estimator for d=2 but will challenge one's patience for d=3 and further challenge the computer's memory for d=4.

Value

For `infer_logLs`, a data frame containing parameter values and their log likelihoods, and additional information such as attributes providing information about the parameter names and statistics names (not detailed here). These attributes are essential for further inferences.

See Details for the required value of the `methods` called by `infer_logLs`.

See Also

See step (3) of the workflow in the Example on the main `Infusion` documentation page.

`infer_SLik_joint` *Infer a (summary) likelihood surface from a simulation table*

Description

This infers the likelihood surface from a simulation table where each simulated data set is drawn for a distinct (vector-valued) parameter, as is usual for reference tables in ABC. A parameter density is inferred, as well as a joint density of parameters and summary statistics, and the likelihood surface is inferred from these two densities.

Usage

```
infer_SLik_joint(data, stat.obs, logLname = Infusion.getOption("logLname"),
                 Simulate = attr(data, "Simulate"),
                 nbCluster= seq_nbCluster(nr=nrow(data)),
                 using = Infusion.getOption("mixturing"),
                 verbose = list(most=interactive(),pedantic=FALSE,final=FALSE),
                 marginalize = TRUE)
```

Arguments

`data`	A data frame, whose each row contains a vector of parameters and one realization of the summary statistics for these parameters.
`stat.obs`	Named numeric vector of observed values of summary statistics.
`logLname`	The name to be given to the log Likelihood in the return object, or the root of the latter name in case of conflict with other names in this object.
`Simulate`	Either NULL or the name of the simulation function if it can be called from the R session.
`nbCluster`	controls the `nbCluster` argument of `Rmixmod::mixmodCluster` ; a vector of integers, or `"max"` which is interpreted as the maximum of the default `nbCluster` value.
`using`	Either `"Rmixmod"` or `"mclust"` to select the clustering methods used.
`marginalize`	Boolean; whether to derive the clustering of fitted parameters by marginalization of the joint clustering (default, and introduced in version 1.3.5); or by a distinct call to a clustering function.
`verbose`	A list as shown by the default, or simply a vector of booleans, indicating respectively whether to display (1) some information about progress; (2) more information whose importance is not clear to me; (3) a final summary of the results after all elements of `simuls` have been processed.

Value

An object of class `SLik_j`, which is a list including an `Rmixmod::mixmodCluster` object (or equivalent objects produced by non-default methods), and additional members not documented here. If projection was used, the list includes a data.frame `raw_data` of cumulated unprojected simulations.

Examples

```
if (Infusion.getOption("example_maxtime")>50) {
  myrnorm <- function(mu,s2,sample.size) {
    s <- rnorm(n=sample.size,mean=mu,sd=sqrt(s2))
    return(c(mean=mean(s),var=var(s)))
  } # simulate means and variances of normal samples of size 'sample.size'
  set.seed(123)
  # pseudo-sample with stands for the actual data to be analyzed:
  ssize <- 40
  Sobs <- myrnorm(mu=4,s2=1,sample.size=ssize)
  # Uniform sampling in parameter space:
```

```
  npoints <- 600
  parsp <- data.frame(mu=runif(npoints,min=2.8,max=5.2),
                      s2=runif(npoints,min=0.4,max=2.4),sample.size=ssize)
  # Build simulation table:
  simuls <- add_reftable(Simulate="myrnorm",par.grid=parsp)
  # Infer surface:
  densv <- infer_SLik_joint(simuls,stat.obs=Sobs)
  # Usual workflow using inferred surface:
  slik_j <- MSL(densv) ## find the maximum of the log-likelihood surface
  slik_j <- refine(slik_j,maxit=5)
  plot(slik_j)
  # etc:
  profile(slik_j,c(mu=4)) ## profile summary logL for given parameter value
  confint(slik_j,"mu") ## compute 1D confidence interval for given parameter
  plot1Dprof(slik_j,pars="s2",gridSteps=40) ## 1D profile
}
```

infer_surface — *Infer a (summary) likelihood or tail probability surface from inferred likelihoods*

Description

The logLs method uses a standard smoothing method (prediction under linear mixed models, a.k.a. Kriging) to infer a likelihood surface, using as input likelihood values themselves inferred with some error for different parameter values. The tailp method use a similar approach for smoothing binomial response data, using the algorithms implemented in the spaMM package for fitting GLMMs with autocorrelated random effects.

Usage

```
## S3 method for class 'logLs'
infer_surface(object, method="REML",verbose=interactive(),allFix=NULL,...)
## S3 method for class 'tailp'
infer_surface(object, method="PQL",verbose=interactive(),allFix,...)
```

Arguments

object	A data frame with attributes, containing independent prediction of logL or of LR tail probabilities for different parameter points, as produced by infer_logLs or infer_tailp.
method	methods used to estimate the smoothing parameters. If method="GCV", a generalized cross-validation procedure is used (for logLs method only). Other methods are as described in the HLfit documentation.
verbose	Whether to display some information about progress or not.
allFix	Fixed values in the estimation of smoothing parameters. For development purposes, not for routine use. For infer_surface.logLs, this should typically include values of all parameters fitted by spaMM::corrHLfit (ρ, ν, ϕ, λ, and \$etaFix=β).

`...`	further arguments passed to or from other methods (currently not used).

Value

An object of class `SLik` or `SLikp`, which is a list including an `HLfit` object as returned by `corrHLfit`, and additional members not documented here.

Examples

```
## see main documentation page for the package
```

`Infusion` *Inference using simulation*

Description

Implements a collection of methods to perform inferences based on simulation of realizations of the model considered. In particular it implements “summary likelihood”, an approach that effectively evaluates and uses the likelihood of simulated summary statistics.

Details

The methods implemented in `Infusion` by default assume that the summary statistics have densities. Special values of some statistic, having discrete probability mass, can be handled using the `boundaries` attribute of the observed summary statistics (see `handling_NAs` for a further use of this attribute).

Note

See examples `example_reftable` for the most complete example using up-to-date workflow, and `example_raw_proj` or `example_raw` for older workflows.

Examples

```
  ## see Note for links to examples.
```

init_reftable *Define starting points in parameter space.*

Description

These functions sample the space of estimated parameters, and also handle other fixed arguments that need to be passed to the function simulating the summary statistics (sample size is likely to be one such argument). The current sampling strategy of these functions is crude but achieves desirable effects for present applications: it samples the space more uniformly, by generating fewer pairs of close points than independent sampling of each point would; it is not exactly a regular grid; and `init_grid` generates replicates of a few parameter points, which were required in the primitive workflow for good smoothing of the likelihood surface. `init_reftable` is a trivial wrapper around `init_grid`, setting the number of replicates to zero, which is appropriate in up-to-date workflows.

Usage

```
init_reftable(lower=c(par=0), upper=c(par=1), steps=NULL,
          nUnique=NULL, maxmin=TRUE, jitterFac=0.5)
init_grid(lower=c(par=0), upper=c(par=1), steps=NULL, nUnique=NULL,
          nRepl=min(10L,nUnique), maxmin=TRUE, jitterFac=0.5)
```

Arguments

`lower`	A vector of lower bounds for the parameters, as well as fixed arguments to be passed to the function simulating the summary statistics. Elements must be named. Fixed parameters character strings.
`upper`	A vector of upper bounds for the parameters, as well as fixed parameters. Elements must be named and match those of `lower`.
`steps`	Number of steps of the grid, in each dimension of estimated parameters. If NULL, a default value is defined from the other arguments. If a single value is given, it is applied to all dimensions. Otherwise, this must have the same length as `lower` and `upper` and named in the same way as the variable parameters in these arguments.
`nUnique`	Number of distinct values of parameter vectors in output. Default is an heuristic guess for good start from not too many points, computed as `floor(50^((v/3)^(1/3)))` where `v` is the number of variable parameters.
`nRepl`	Number of replicates of distinct values of parameter vectors in output.
`maxmin`	Boolean. If TRUE, use a greedy max-min strategy (GMM, inspired from Ravi et al. 1994) in the selection of points from a larger set of points generated by an hypercube-sampling step. If FALSE, `sample` is instead used for this second step. This may be useful as the default method becomes slow when thousands of points are to be sampled.
`jitterFac`	Controls the amount of jitter of the points around regular grid nodes. The default value 0.5 means that a mode can move by up to half a grid step (independently in each dimension), so that two adjacent nodes moved toward each other can (almost) meet each other.

Value

A data frame. Each row defines a list of arguments of vector of the function simulating the summary statistics.

Note

`init_grid` is an exported function from the **blackbox** package.

References

Ravi S.S., Rosenkrantz D.J., Tayi G.K. 1994. Heuristic and special case algorithms for dispersion problems. Operations Research 42, 299-310.

Examples

```
set.seed(123)
init_grid()
init_grid(lower=c(mu=2.8,s2=0.5,sample.size=20),
          upper=c(mu=5.2,s2=4.5,sample.size=20),
          steps=c(mu=7,s2=9),nUnique=63)
```

MSL *Maximum likelihood from an inferred likelihood surface*

Description

This computes the maximum of an object of class `SLik` representing an inferred (summary) likelihood surface

Usage

```
MSL(object, CIs = TRUE, level = 0.95, verbose = interactive(),
    eval_RMSEs = TRUE, cluster_args=list(),init=NULL, prior_logL=NULL,
    ...)
```

Arguments

`object`	an object of class `SLik_j` as produced by `infer_SLik_joint` (or, in the primitive workflow, of class `SLik` as produced by `infer_surface.logLs`).
`CIs`	If `TRUE`, construct one-dimensional confidence intervals for all parameters.
`level`	Intended coverage probability of the confidence intervals.
`verbose`	Whether to display some information about progress and results.
`eval_RMSEs`	Logical: whether to evaluate prediction uncertainty for likelihoods/ likelihood ratios/ parameters.

`cluster_args`	A list of arguments, passed to `makeCluster`, to control parallel computation of RMSEs. Beware that parallel computation of RMSEs tends to be memory-intensive. The list may contain a non-null `spec` element, in which case the `nb_cores` global **Infusion** option is ignored. Do ***not*** use a structured list with an `RMSE` element as is possible for `refine` (see Details of `refine` documentation).
`init`	Initial value for the optimiser. Better ignored.
`prior_logL`	(effective only for up-to-date workflow using gaussian mixture modelling of a joint distribution of parameters and statistics) a function that returns a vector of prior log-likelihood values, which is then added to the likelihood deduced from the summary likelihood analysis. The function's single argument must handle a matrix similar to the `newdata` argument of `predict.SLik_j`.
`...`	Further arguments passed from or to other methods.

Details

If Kriging has been used to construct the likelihood surface, `RMSEs` are computed using approximate formulas for prediction (co-)variances in linear mixed midels (see Details in `predict`). Otherwise, a more computer-intensive bootstrap method is used. `par_RMSEs` are computed from `RMSEs` and from the numerical gradient of profile log-likelihood at each CI bound. Only `RMSEs`, not `par_RMSEs`, are compared to `precision`.

Value

The `object` is returned invisibly, with the following added members, each of which being (as from version 1.5.0) an environment:

`MSL` containing variables `MSLE` and `maxlogL` that match the `par` and `value` returned by an `optim` call. Also contain the `hessian` of summary likelihood at its maximum.

`RMSEs` containing, as variable `RMSEs`, the root mean square errors of the log-likelihood at its inferred maximum and of the log-likelihood ratios at the CI bounds.

`par_RMSEs` containing, as variable `par_RMSEs`, root mean square errors of the CI bounds.

To ensure backward-compatibility of code to possible future changes in the structure of the objects, the extractor function `get_from` should be used to extract the `RMSEs` and `par_RMSEs` variables from their respective environments, and more generally to extract any element from the objects.

Examples

```
## see main documentation page for the package
```

multi_binning *Multivariate histogram*

Description

Constructs a multivariate histogram of the points. Optionally, first tests whether a given value is within the convex hull of input points and constructs the histogram only if this test is TRUE. This function is available for development purposes but is not required otherwise . It is sparsely documented and subject to changes without notice.

Usage

```
multi_binning(m, subsize=trunc(nrow(m)^(Infusion.getOption("binningExponent"))),
              expand=5/100, focal=NULL)
```

Arguments

`m`	A matrix representing points in d-dimensional space, where d is the number of columns
`subsize`	A control parameter for an undocumented algorithm
`expand`	A control parameter for an undocumented algorithm
`focal`	Value to be tested for inclusion within the convex hull. Its elements must have names.

Details

The algorithm may be detailed later.

Value

Either NULL (if the optional test returned FALSE), or an histogram represented as a data frame each row of which represents an histogram cell by its barycenter (a point in d-dimensional space), its “binFactor” (the volume of the cell times the total number of observations) and its “count” (the number of observations within the cell). The returned data frame has the following attributes: `attr(.,"stats")` are the column names of the d-dimensional points; `attr(.,"count")` is the column name of the count, and `attr(.,"binFactor")` is the column name of the binFactor.

`options` *Infusion options settings*

Description

Allow the user to set and examine a variety of *options* which affect operations of the Infusion package. However, typically these should not be modified, and if they are, not more than once in a data analysis.

Usage

```
Infusion.options(...)

Infusion.getOption(x)
```

Arguments

`x`	a character string holding an option name.
`...`	A named value or a list of named values. The following values, with their defaults, are used in `Infusion`:

- **`mixturing`** character string: package or function to be used for mixture modelling. Recognized packages are `"Rmixmod"` (the default) and `"mclust"`;
- **`train_cP_size`:** Expression for `train_cP_size` argument of `project.character`.
- **`trainingsize`:** Expression for `trainingsize` argument of `project.character`.
- **`projKnotNbr = 1000`:** default value of `trainingsize` argument of `project.character` for REML (as implied by default expression for `trainingsize`).
- **`logLname = "logL"`:** default value of `logLname` argument of `infer_logLs`. The name given to the inferred log likelihoods in all analyses.
- **`LRthreshold= - qchisq(0.999,df=1)/2`:** A value used internally by `sample_volume` to sample points in the upper region of the likelihood surface, as defined by the given likelihood ratio threshold.
- **`precision = 0.1`:** default value of `precision` argument of `refine`. Targets RMSE of log L and log LR estimates.
- **`nRealizations=1000`:** default value of `nRealizations` argument of `add_simulation`. Number of realizations for each empirical distribution.
- **`mixmodGaussianModel="Gaussian_pk_Lk_Dk_A_Dk"`:** default models used in clustering by `Rmixmod`. Run `Rmixmod::mixmodGaussianModel()` for a list of possible models, and see the statistical documentation (Mixmod Team 2016) for explanations about them.
- **`seq_nbCluster= function(projdata, nr=nrow(projdata)) {seq(ceiling(nr^0.3))}`:** function to control the value of `nbCluster` used in clustering by `Rmixmod` (see Details for discussion of this default).
- **`maxnbCluster = function(projdata) {...}`:** function to control the maximum number of clusters (see Details).

`example_maxtime=2.5`**:** Used in the documentation to control whether the longer examples should be run. The approximate running time of given examples (or some very rough approximation for it) on one author's laptop is compared to this value.

`nb_cores` Number of cores for parallel computations (see Details for implementation of these).

`gof_nstats_fn` See `goftest`.

and possibly other undocumented values for development purposes.

Details

The set of the number of clusters tried (`nbCluster` argument in `Rmixmod`) is controlled by two options: `seq_nbCluster` and `maxnbCluster`. The second is used to correct the first, using the dimensions of the `projdata` locally used for clustering, which typically differs from the dimensions of the user-level `data` (if projections have been applied, in particular). The default upper value of the `nbCluster` range is the value recommended in the `mixmod` statistical documentation (Mixmod Team, 2016). But this default may be suitable only for low-dimensional data, hence the need for correcting it by`maxnbCluster`.

`Infusion` can perform parallel computations if several cores are available and requested though `Infusion.options(nb_cores=.)`. If the `doSNOW` back-end is attached (by explicit request from the user), it will be used; otherwise, `pbapply` will be used. Both provide progress bars, but `doSNOW` may provide more efficient load-balancing. The character shown in the progress bar is `'P'` for parallel via `doSNOW` backend, `'p'` for parallel via `pbapply` functions, and `'s'` for serial via `pbapply` functions. In addition, `add_simulation` can parallelise at two levels: at an outer level over parameter point, or at an inner level over simulation replicates for each parameter point. The progress bar of the outer computation is shown, but the character shown in the progress bar is `'N'` if the inner computation is parallel via the `doSNOW` backend, and `'n'` if it is parallel via `pbapply` functions. So, one should see either `'P'` or `'N'` when using `doSNOW`.

Value

For `Infusion.getOption`, the current value set for option `x`, or `NULL` if the option is unset.

For `Infusion.options()`, a list of all set options. For `Infusion.options(name)`, a list of length one containing the set value, or `NULL` if it is unset. For uses setting one or more options, a list with the previous values of the options changed (returned invisibly).

References

Mixmod Team (2016). Mixmod Statistical Documentation. Université de Franche-Comté, Besançon, France. Version: February 10, 2016 retrieved from `https://www.mixmod.org`.

Examples

```
  Infusion.options()
  Infusion.getOption("LRthreshold")
  ## Not run:
  Infusion.options(LRthreshold=- qchisq(0.99,df=1)/2)

## End(Not run)
```

plot.SLik — *Plot SLik or SLikp objects*

Description

Mostly conceived for exposition purposes, for the two-parameters case. The black-filled points are those for which the observed summary statistic was outside of the convex hull of the simulated empirical distribution. The crosses mark the estimated ML point and the confidence intervals points, that is, the outmost points on the contour defined by the profile likelihood threshold for the profile confidence intervals. There is a pair of CI points for each interval. The smaller black dots mark points added in the latest iteration, if `refine` was used.

Usage

```
## S3 method for class 'SLik'
plot(x, y, filled = FALSE, decorations = NULL,
                    color.palette = NULL, plot.axes = NULL,
                    plot.title = NULL, plot.slices=TRUE, ...)
## S3 method for class 'SLik_j'
plot(x, y, filled = nrow(x$logLs)>5000L, decorations = NULL,
                      color.palette = NULL, plot.axes = NULL,
                   plot.title = NULL, from_refine=FALSE, plot.slices=TRUE, ...)
```

Arguments

`x`	An object of class `SLik` or `SLikp`
`y`	Not used, but included for consistency with the `plot` generic.
`filled`	whether to plot a `mapMM` or a `filled.mapMM`.
`decorations`	Graphic directives added to the default `decorations` value in calls of `mapMM` or a `filled.mapMM` (see the source code of `plot.SLik` for the latter default values).
`color.palette`	Either NULL or a function that can replace the default color function used by `plot.SLik`. The function must have a single argument, giving the number of color levels.
`plot.title`	statements which replace the default titles to the main plot (see Details).
`plot.axes`	statements which replace the default axes on the main plot (see Details).
`from_refine`	For programming purposes, not documented.
`plot.slices`	boolean: whether to plot “slices” of the summary-likelihood surface for pairs of parameters (p1,p2), when more than two parameters are fitted. In such plots the additional parameters p3, p4... are fixed to their estimates [in contrast to profile plots where p3, p4... take distinct values for each (p1,p2), maximizing the function for each (p1,p2)].
`...`	further arguments passed to or from other methods (currently can be used to pass a few arguments such as `map.asp` in all cases, or `variances` to `filled.mapMM`).

Details

Different graphic functions are called depending on the number of estimated parameters. For two parameters, `mapMM` or `filled.mapMM` are called. For more than two parameters, `spaMM.filled.contour` is called. See the documentation of these functions for the appropriate format of the `plot.title` and `plot.axes` arguments.

Value

Returns the plotted object invisibly.

Examples

```
## Not run:
## Using 'slik_j' object from the example in help("example_reftable")
plot(slik_j,filled=TRUE,
     plot.title=quote(title("Summary-likelihood-ratio surface",
                            xlab=expression(mu),
                            ylab=expression(sigma^2))))

## End(Not run)
```

`plot1Dprof` *Plot likelihood profiles*

Description

These functions plot 1D and 2D profiles from a summary-likelihood object.

High quality 2D plots may be slow to compute, and there may be many of them in high-dimensional parameter spaces, so parallelization of the computation of each profile point has been implemented for them. Usual caveats apply: there is an time cost of launching processes on a cluster, particularly on socket clusters, possibly offsetting the benefits of parallelization when each profile point is fast to evaluate. Further, summary-likelihood objects are typically big (memory-wise), which may constrain the number of concurrent processes.

Usage

```
plot1Dprof(object, pars=object$colTypes$fittedPars, type="logLR",
           gridSteps=21, xlabs=list(), ylab, scales=NULL,
           plotpar=list(pch=20),
           control=list(min=-7.568353, shadow_col="grey70"),
           decorations = function(par) NULL, ...)
plot2Dprof(object, pars=object$colTypes$fittedPars, type="logLR",
           gridSteps=17, xylabs=list(), main, scales=NULL,
           plotpar=list(pch=20), margefrac = 0,
           decorations = function(par1,par2) NULL,
           filled.contour.fn = "spaMM.filled.contour",
           cluster_args=NULL, ... )
```

Arguments

`object`	An `SLik` or `SLik_j` object
`pars`	Control of parameters for which profiles will be computed. If `pars` is specified as a vector of names, profiles are plotted for each parameter, or (2D case) for all pairs of distinct parameters. Finer control is possible in the 2D case (see Details).
`type`	Character: `"logL"` to plot the log-likelihood profile; `"logLR"` (or `"LR"` for the not-log version) to plot the log-likelihood-ratio profile (the default); or `"zoom"` or `"dual"` for variants of `"logLR"` (see details).
`gridSteps`	The number of values (in each dimension for 2D plots) which likelihood should be computed. For 1D plots, `gridSteps=0` is now obsolete.
`xlabs`	A *list* of alternative axis labels. The names of the list elements should be elements of `pars` (see Examples)
`xylabs`	Same as `xlabs` but affecting both axes in 2D plots
`ylab`	Same as `ylab` argument of `plot`. Default depends on `type` argument.
`main`	Same as `main` argument of `plot`. Default depends on `type` argument.
`scales`	A named character vector, which controls ticks and tick labels on axes, so that these can be expressed as (say) the exponential of the parameter inferred in the SLik object. For example if the likelihood of `logPop` = log(population size) was thus inferred, `scales=c(logPop="log")` will give population size values on the axis (but will retain a log scale for this parameter). Possible values of each element of the vector are `"identity"` (default), `"log"`, and `"log10"`,
`plotpar`	Arguments for `par()` such as font sizes, etc.
`control`	Control of `"zoom"` or `"dual"` plots (see Details).
`decorations`	A function with formals parameters as shown by the default (being parameters names represented as character strings), implementing graphic directives added to the plot.
`margefrac`	For development purposes, not documented.
`filled.contour.fn`	Name of a possible alternative to `graphics::filled.contour` to be used for rendering the plot.
`cluster_args`	NULL, or a list in which case a cluster may be created and used. The list elements must match the arguments `spec` and `type` of `parallel::makeCluster`. A socket cluster is created unless `type="FORK"` (on operating systems that support fork clusters).
`...`	Further arguments passed by another function. Currently these arguments are ignored.

Details

In the 2D case, the `pars` may be specified as a two-column natrix, in which case profiles are generated for all pairs of distinct parameters specified by rows of the matrix. It may also be specified as a two-element list, where each element is a vector of parameter names. In that case, profiles are generated for all pairs of distinct parameters combining one element of each vector.

A "zoom" plot shows only the top part of the profile, defined as points whose y-values are above a threshold minus-log-likelihood ratio `control$min`, whose default is -7.568353, the 0.9999 p-value threshold.

A "dual" plot displays both the zoom, and a shadow graph showing the full profile. The dual plot is shown only when requested and if there are values above and below `control$min`. The shadow curve color is given by `control$shadow_col`.

Value

Both functions return a list, which currently has a single element `MSL_updated` which is a boolean indicating whether the summary-likelihood maximum (but not the intervals) has been recomputed.

Examples

```
if (Infusion.getOption("example_maxtime")>20) {
  #### Toy bivariate gaussian model, three parameters, no projections
  #
  myrnorm2 <- function(mu1,mu2,s2,sample.size) {
    sam1 <- rnorm(n=sample.size,mean=mu1,sd=sqrt(s2))
    sam2 <- rnorm(n=sample.size,mean=mu2,sd=sqrt(s2))
    s <- c(sam1,sam2)
    e_mu <- mean(s)
    e_s2 <- var(s)
    c(mean=e_mu,var=e_s2,kurt=sum((s-e_mu)^4)/e_s2^2)
  }
  #
  ## pseudo-sample, standing for the actual data to be analyzed:
  set.seed(123)
  Sobs <- myrnorm2(mu1=4,mu2=2,s2=1,sample.size=40) ##
  #
  ## build reference table
  parsp <- init_reftable(lower=c(mu1=2.8,mu2=1,s2=0.2),
                         upper=c(mu1=5.2,mu2=3,s2=3),
                         nUnique=600)
  parsp <- cbind(parsp,sample.size=40)
  simuls <- add_reftable(Simulate="myrnorm2",par.grid=parsp)

  ## Inferring the summary-likelihood surface...
  densv <- infer_SLik_joint(simuls,stat.obs=Sobs)
  slik_j <- MSL(densv) ## find the maximum of the log-likelihood surface

  ## plots
  plot2Dprof(slik_j,gridSteps=21,
             ## alternative syntaxes for non-default 'pars':
             # pars = c("mu1","mu2"), # => all combinations of given elements
             # pars = list("s2",c("mu1","mu2")), # => combinations via expand.grid()
             # pars = matrix(c("mu1","mu2","s2","mu1"), ncol=2), # => each row of matrix
             xylabs=list(
               mu1=expression(paste(mu[1])),
               mu2=expression(paste(mu[2])),
               s2=expression(paste(sigma^2))
```

```
              ))
  # One could also add (e.g.)
  #            cluster_args=list(spec=4, type="FORK"),
  # when longer computations are requested.
}

if (Infusion.getOption("example_maxtime")>40) {
 #### Older example with primitive workflow
 data(densv)
 slik <- infer_surface(densv) ## infer a log-likelihood surface
 slik <- MSL(slik) ## find the maximum of the log-likelihood surface
 plot1Dprof(slik,pars="s2",gridSteps=40,xlabs=list(s2=expression(paste(sigma^2))))
}
```

`predict.SLik_j` *Evaluate log-likelihood for given parameters*

Description

As the Title says. Implemented as a method of the `predict` generic, for objects created by the up-to-date workflow using gaussian mixture modelling of a joint distribution of parameters and statistics (hence the `newdata` argument, shared by many `predict` methods; but these `newdata` should be parameter values, not data).

Usage

```
## S3 method for class 'SLik_j'
predict(
  object, newdata, log = TRUE, which = "lik",
  tstat = t(attr(object$logLs, "stat.obs")),
  solve_t_chol_sigma_lists = object$clu_params$solve_t_chol_sigma_lists,
  ...)
```

Arguments

`object`	an object of class `SLik_j`, as produced by `infer_SLik_joint`.
`newdata`	A matrix, whose rows each contain a full vector of the fitted parameters; or a single vector. If parameter names are not provided (as column names in the matrix case), then the vector is assumed to be ordered as `object$colTypes$fittedPars`.
`log`	Boolean: whether to return log-likelihood or likelihood.
`which`	`"lik"` or `"safe"`. The latter protects against some artefacts of predictions beyond the regions of parameter space well sampled by the inference procedure.
`tstat`	The data (as projected summary statistics). Defaults to the data input in the inference procedure (i.e., the projected statistics used as `stat.obs` argument of `infer_SLik_joint`).
`solve_t_chol_sigma_lists`	For programming purposes. Do not change this argument.
`...`	For consistency with the generic. Currently ignored.

Value

Numeric: a single value, or a vector of (log-)likelihoods for different rows of the input `newdata`.

Examples

```
## see help("example_reftable")
```

`profile.SLik` *Compute profile summary likelihood*

Description

Predicts the profile likelihood for a given parameter value (or vector of such values) using predictions from an `SLik` object (as produced by `MSL`).

Usage

```
## S3 method for class 'SLik'
profile(fitted, value, fixed=NULL, return.optim=FALSE,
                        init = "default", which="safe", ...)
## S3 method for class 'SLik_j'
profile(fitted, ...)
```

Arguments

`fitted`	an `SLik` object.
`value`	The parameter value (as a vector of named values) for which the profile is to be computed
`fixed`	When this is `NULL` the computed interval is a profile confidence interval over all parameters excluding `value`. `fixed` allows one to set fixed values to some of these parameters.
`return.optim`	If this is TRUE, and if maximization of likelihood given `value` and `fixed` is indeed required, then the full result of the optimization call is returned.
`...`	For `SLik_j` method, arguments passed to `SLik` method. For `SLik_j` method, currently not used.
`init`	Better ignored. Either a named vector of parameter values (initial value for some optimizations) or a character string. The default is to call a procedure to find a good initial point from a set of candidates. The source code should be consulted for further details and is subject to change without notice.
`which`	Better ignored (for development purpose).

Value

The predicted summary profile log-likelihood; or possibly the result of an optimization call if `return.optim` is TRUE.

Examples

```
## see main documentation page for the package
```

---

project.character *Learn a projection method for statistics and apply it*

Description

project is a generic function with two methods. If the first argument is a parameter name, project.character (alias: get_projector) defines a projection function from several statistics to an output statistic predicting this parameter. project.default (alias: get_projection) produces a vector of projected statistics using such a projection. project is particularly useful to reduce a large number of summary statistics to a vector of projected summary statistics, with as many elements as parameters to infer. This dimension reduction can substantially speed up subsequent computations. The concept implemented in project is to fit a parameter to the various statistics available, using machine-learning or mixed-model prediction methods. All such methods can be seen as nonlinear projection to a one-dimensional space. project.character is an interface that allows different projection methods to be used, provided they return an object of a class that has a defined predict method with a newdata argument (as expected, see predict).

plot_proj is a hastily written convenience function to plot a diagnostic plot for a projection from an object of class SLik_j.

Usage

```
project(x,...)

## S3 method for building the projection
## S3 method for class 'character'
project(x, stats, data,
             trainingsize=  eval(Infusion.getOption("trainingsize")),
             train_cP_size= eval(Infusion.getOption("train_cP_size")),
             method, methodArgs=list(), verbose=TRUE,...)
get_projector(...) # alias for project.character

## S3 method for applying the projection
## Default S3 method:
project(x, projectors, use_oob=Infusion.getOption("use_oob"),
                         is_trainset=FALSE, methodArgs=list(), ...)
get_projection(...) # alias for project.default

plot_proj(object, parm, proj, xlab=parm, ylab=proj, ...)
```

Arguments

`x`	The name of the parameter to be predicted, or a vector/matrix/list of matrices of summary statistics.
`stats`	Statistics from which the predictor is to be predicted
`use_oob`	Boolean: whether to use out-of-bag predictions for data used in the training set, when such oob predictions are available (i.e. for random forest methods). Default as controlled by the same-named package option, is TRUE. This by default involves a costly check on each row of the input `x`, whetehr it belongs to the training set, so it is better to set it to FALSE if you are sure `x` does not belong to the training set (for true data in particular). Alternatively the check can be bypassed if you are sure that `x` was used as the training set.
`is_trainset`	Boolean. Set it to TRUE if `x` was used as the training set, to bypass a costly check (see `use_oob` argument).
`data`	A list of simulated empirical distributions, as produced by `add_simulation`, or a data frame with all required variables.
`trainingsize, train_cP_size`	Integers; for most projection methods (excluding `"REML"` but including `"ranger"`) only `trainingsize` is taken into account: it gives the maximum size of the training set (and is infinite by default for `"ranger"` method). If the `data` have more rows the training set is randomly sampled from it. For the `"REML"` method, `train_cP_size` is the maximum size of the data used for estimation of smoothing parameters, and `trainingsize` is the maximum size of the data from which the predictor is built given the smoothing parameters.
`method`	character string: `"REML"`, `"GCV"`, or the name of a suitable projection function. The latter may be defined in another package, e.g. `"ranger"` or `"randomForest"`, or predefined by `Infusion`, or defined by the user. See Details for predefined functions and for defining new ones. The default method is `"ranger"` if this package is installed, and `"REML"` otherwise. Defaults may change in later versions, so it is advised to provide an explicit `method` to improve reproducibility.
`methodArgs`	A list of arguments for the projection method. For `project.character`, the `ranger` method is run with some default argument if no `methodArgs` are specified. Beware that a NULL `methodArgs$splitrule` is interpreted as the `"extratrees"` `splitrule`, whereas in a direct call to `ranger`, this would be interpreted as the `"variance"` `splitrule`. For `project.default`, the only `methodArgs` element handled is `num.threads` passed to `predict.ranger` (which can also be controlled globally by `Infusion.options(nb_cores=.)`). For other methods, `project` kindly (tries to) assign values to the required arguments if they are absent from `methodArgs`, according to the following rules: If `"REML"` or `"GCV"` methods are used (in which case `methodArgs` is completely ignored); or if the projection method uses `formula` and `data` arguments (in particular if the formula is of the form `response ~ var1 + var2 + ...`; otherwise the formula should be provided through `methodArgs`). This works for example for methods based on `nnet`; or

	if the projection method uses `x` and `y` arguments. This works for example for the (somewhat obsolete) method `randomForest` (though not with the generic function `method="randomForest"`, but only with the internal function `method="randomForest:::randomFo`
`projectors`	A list with elements of the form `<name>=<project result>`, where the `<name>` must differ from any name of `x`. `<project result>` may indeed be the return object of a `project` call.
`verbose`	Whether to print some information or not. In particular, `TRUE`, true-vs.-predicted diagnostic plots will be drawn for projection methods "known" by Infusion (notably `"ranger"`, `"fastai.tabular.learner.TabularLearner"`, `"keras::keras.engine.training.M` `"randomForest"`, `"GCV"`, `caret::train`).
`object`	An object of class `SLik_j`.
`parm`	Character string: a parameter name.
`proj`	Character string: name of projected statistic.
`xlab,ylab`	Passed to `plot`.
`...`	Further arguments passed to or from other functions. For `plot_proj`, they are passed to `plot`.

Details

The preferred `project` method is non-parametric regression by (variants of) the random forest method as implemented in **ranger**. It is the default method, if that package is installed. Alternative methods have been interfaced as detailed below, but the functionality of most interfaces is infrequently tested.

By default, the ranger call through `project` will use the split rule `"extratrees"`, with some other controls also differing from the **ranger** package defaults. If the split rule `"variance"` is used, the default value of `mtry` used in the call is also distinct from the **ranger** default, but consistent with Breiman 2001 for regression tasks.

Machine learning methods such as random forests overfit, *except if* out-of-bag predictions are used. When they are not, the bias is manifest in the fact that using the same simulation table for learning the projectors and for other steps of the analyses tend to lead to too narrow confidence regions. This bias disappears over iterations of `refine` when the projectors are kept constant. `Infusion` avoid this bias by using out-of-bag predictions when relevant, when `ranger` and `randomForest` are used. But it provides no code handling that problem for other machine-learning methods. Then, users should cope with that problems, and at a minimum should not update projectors in every iteration (the "Gentle Introduction to Infusion may contain further information about this problem").

Prediction can be based on a linear mixed model (LMM) with autocorrelated random effects, internally calling the `corrHLfit` function with formula `<parameter> ~ 1+ Matern(1|<stat1>+...+<statn>)`. This approach allows in principle to produce arbitrarily complex predictors (given sufficient input) and avoids overfitting in the same way as restricted likelihood methods avoids overfitting in LMM. REML methods are then used by default to estimate the smoothing parameters. However, faster methods are generally required.

To keep REML computation reasonably fast, the `train_cP_size` and `trainingsize` arguments determine respectively the size of the subset used to estimate the smoothing parameters and the size of the subset defining the predictor given the smoothing parameters. REML fitting is already slow for data sets of this size (particularly as the number of predictor variables increase).

If `method="GCV"`, a generalized cross-validation procedure (Golub et al. 1979) is used to estimate the smoothing parameters. This is faster but still slow, so a random subset of size `trainingsize` is still used to estimate the smoothing parameters and generate the predictor.

Alternatively, various machine-learning methods can be used (see e.g. Hastie et al., 2009, for an introduction). A random subset of size `trainingsize` is again used, with a larger default value bearing the assumption that these methods are faster. Predefined methods include

- `"ranger"`, the default, a computationally efficient implementation of random forest;
- `"randomForest"`, the older default, probably obsolete now;
- `"neuralNet"`, a neural network method, using the `train` function from the `caret` package (probably obsolete too);
- `"fastai"` deep learning using the `fastai` package;
- `"keras"` deep learning using the `keras` package.

The last two interfaces may yet offer limited or undocumented control: using deep learning seems attractive but the benefits over `"ranger"` are not clear (notably, the latter provide out-of-bag predictions that avoid overfitting).

In principle, any object suitable for prediction could be used as one of the `projectors`, and `Infusion` implements their usage so that in principle unforeseen projectors could be used. That is, if predictions of a parameter can be performed using an object `MyProjector` of class `MyProjectorClass`, `MyProjector` could be used in place of a `project` result if `predict.MyProjectorClass(object,newdata,...)` is defined. However, there is no guarantee that this will work on unforeseen projection methods, as each method tends to have some syntactic idiosyncrasies. For example, if the learning method that generated the projector used a formula-data syntax, then its `predict` method is likely to request names for its `newdata`, that need to be provided through `attr(MyProjector,"stats")` (these names cannot be assumed to be in the `newdata` when `predict` is called through `optim`).

Value

`project.character` returns an object of class returned by the `method` (methods `"REML"` and `"GCV"` will call `corrHLfit` which returns an object of class spaMM) `project.default` returns an object of the same class and structure as the input `x`, containing the projected statistics inferred from the input summary statistics.

Note

See workflow examples in `example_reftable` and `example_raw_proj`.

References

Breiman, L. (2001). Random forests. Mach Learn, 45:5-32. <doi:10.1023/A:1010933404324>

Golub, G. H., Heath, M. and Wahba, G. (1979) Generalized Cross-Validation as a method for choosing a good ridge parameter. Technometrics 21: 215-223.

T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer, New York, 2nd edition, 2009.

Examples

```
## see Note for links to examples.
```

`refine` *Refine estimates iteratively.*

Description

This is a generic function with currently methods for `SLik`, `SLik_j` and `SLikp` objects (as produced by `MSL`). Depending on the value of its `newsimuls` argument, and on whether the function used to generate empirical distributions can be called by R, it (1) defines new parameters points and/or (2) infers their summary likelihood or tail probabilities for each parameter point independently, adds the inferred values results as input for refined inference of likelihood or P-value response surface, and provides new point estimates and confidence intervals.

Usage

```
## S3 method for class 'SLik'
refine(object, method=NULL, ...)


## Default S3 method:
refine(object, surfaceData, Simulate =
            attr(surfaceData,"Simulate"), maxit = 1, n = NULL,
            useEI = list(max=TRUE,profileCI=TRUE,rawCI=FALSE),
         newsimuls = NULL, trypoints=NULL, CIs = useCI, useCI = TRUE, level = 0.95,
          verbose = list(most=interactive(),final=NULL,movie=FALSE,proj=FALSE),
            precision = Infusion.getOption("precision"),
            nb_cores = NULL, packages=attr(object$logLs,"packages"),
            env=attr(object$logLs,"env"), method,  using = object$using,
            eval_RMSEs=TRUE, update_projectors = FALSE,
            cluster_args=list(),
            cl_seed=.update_seed(object),
            nbCluster=quote(refine_nbCluster(nr=nrow(data))),
            ...)
```

Arguments

`object`	an `SLik` or `SLik_j` object
`surfaceData`	A data.frame with attributes, usually taken from the `object` and thus **not** specified by user, usable as input for `infer_surface`.
`Simulate`	Character string: name of the function used to simulate samples. The only meaningful non-default value is `NULL`, in which case `refine` may return (if `newsimuls` is also `NULL`) a data frame of parameter points on which to run a simulation function.
`maxit`	Maximum number of iterative refinements (see also `precision` argument)

`n`	NULL or numeric, for a number of parameter points (excluding replicates and confidence interval points in the primitive workflow), whose likelihood should be computed (see `n` argument of `sample_volume`). This argument is typically not heeded in the first refinement iteration (only one fifth as many points may be produced), but will be closely approached in later ones (so four refinement iterations with `n=1000` is expected to produce 3200 new points). If `n` is left NULL, the number of points of the initial reference table is used as a reference, but with a somewhat different effect: four refinement iterations starting from a reference table of `1000` ones iis expected to produce 4000 new points (though again, possibly only 200 in the first refinement iteration).
`useEI`	Cf this argument in `rparam`
`newsimuls`	For the `SLik_j` method, a matrix or data frame, with the same parameters and summary statistics as the `data` of the original `infer_SLik_joint` call. For other methods, a `list` of simulation of distributions of summary statistics, in the same format as for `link{add_simulation}`. If no such list is provided (i.e., if `newsimuls` remains `NULL`), the `attr(object$logLs,"Simulate")` function is used (it is inherited from the `Simulate` argument of `add_simulation` through the initial sequence of calls of functions `add_simulation`, `infer_logLs` or `infer_tailp`, and `infer_surface`). If no such function is available, then this function returns parameters for which new distribution should be provided by the user.
`trypoints`	A data frame of parameters on which the simulation function `attr(object$logLs,"Simulate")` should be called to extend the reference table. Only for programming by expert users, because poorly thought input `trypoints` could severely affect the inferences.
`CIs`	Boolean: whether to infer bounds of (one-dimensional, profile) confidence intervals. Their computation is not quite reliable in parameter spaces of large dimensions, so they should not be trusted per se, yet they may be useful for the definition of new parameter points.
`useCI`	whether to include parameter points near the inferred confidence interval points in the set of points whose likelihood should be computed. Effective only if `CIs` was TRUE.
`level`	Intended coverage of confidence intervals
`verbose`	A list as shown by the default, or simply a vector of booleans. `verbose$most` controls whether to display information about progress and results, except plots; `$final` controls whether to `plot()` the final `object` to show the final likelihood surface. Default is to plot it only in an interactive session and if fewer than three parameters are estimated; `$movie` controls whether to `plot()` the updated `object` in each iteration; `verbose$proj` controls the `verbose` argument of `project.character`. If `verbose` is an unnamed vector of booleans, they are matched to as many elements from `"most","movie","final","proj"`, in that order.
`precision`	Requested local precision of surface estimation, in terms of prediction standard errors (RMSEs) of both the maximum summary log-likelihood and the likelihood ratio at any CI bound available. Iterations will stop when either `maxit` is reached, or if the RMSEs have been computed for the object (see `eval_RMSEs`

	argument) and this precision is reached for the RMSEs. A given precision on the CI bounds themselves might seem more interesting, but is not well specified by a single precision parameter if the parameters are on widely different scales.
nb_cores	Shortcut for cluster_args$spec for sample simulation.
cluster_args	A list of arguments for makeCluster, in addition to makeCluster's spec argument which is in most cases best specified by the nb_cores argument. Cluster arguments allow independent control of parallel computations for the different steps of a refine iteration (see Details; as a rough but effective summary, use only nb_cores when the simulations support it, and only cluster_args=list(project=list(num.thre when they do not).
packages	NULL or a list with possible elements add_simulation and logL_method, passed respectively as the packages arguments of add_simulation and infer_logLs, wherein they are the additional packages to be loaded on child processes. The default value keeps pre-refine values over iterations.
env	An environment, passed as the env argument to add_simulation. The default value keeps the pre-refine value over iterations.
using	Passed to infer_SLik_joint: a charcter string used to control the joint-density estimation method, as documented for that function. Default is to use to same method as in the the first iteration, but this argument allows a change of method.
method	(A vector of) suggested method(s) for estimation of smoothing parameters (see method argument of infer_surface), and therefore controlling the primitive workflow (see using instead for controlling the up-to-date workflow). The ith element of the vector is used in the ith iteration, if available; otherwise the last element is used. This argument is not always heeded, in that REML may be used if the suggested method is GCV but it appears to perform poorly. The default for SLikp and SLikp objects are "REML" and "PQL", respectively.
eval_RMSEs	passed to MSL
update_projectors	Boolean; whether to update the projectors at each iteration.
cl_seed	NULL or integer, passed to add_simulation. The default code uses an internal function, .update_seed, to update it from a previous iteration.
nbCluster	Passed to infer_SLik_joint. The data in the expression for the default value refers to the data argument of the latter function.
...	further arguments passed to or from other methods. refine passes these arguments to the plot method suitable for the object.

Details

New parameter points are sampled as follows: the algorithm aims to sample uniformly the space of parameters contained in the confidence regions defined by the level argument, and to surround it by a region sampled proportionally to likelihood. In each iteration the algorithm aims to add as many points (say *n*) as computed in the first iteration, so that after k iterations of refine, there are $n * (k + 1)$ points in the simulation table. However, when not enough points satisfy certain criteria, only *n/5* points may be added in an iteration, this being compensated in further iterations. For example, if $n = 600$, the table may include only 720 points after the first refine, but 1800 after the second.

Independent control of parallelisation may be needed in the different steps, e.g. if the simulations are not easily parallelised whereas the projection method natively handles parallelisation. In the up-to-date workflow with default ranger projection method, prarallelisation controls may be passed to add_reftable for sample simulations, to project methods when projections are updated, and to MSL for RMSE computations (alternatively for the primitive workflow, add_simulation, infer_logLs and MSL are called). nb_cores, if given and not overcome by other options, will control simulation and projection steps (but not RMSE computation): nb_cores gives the number of parallel processes for sample simulation, with additional makeCluster arguments taken from cluster_args, but RMSE computations are performed serially. Further independent control is possible as follows:
cluster_args=list(project=list(num.threads=<.>)) allows control of the num.threads argument of **ranger** functions;
cluster_args=list(RMSE=list(spec=<number of 'children'>)) can be used to force parallel computation of RMSEs;
cluster_args=list(spec=<.>, <other makeCluster arguments>)) would instead apply the same arguments to both reference table and RMSE computation, overcoming the default effect of nb_cores in both of them; finally
cluster_args=list(reftable=list(<makeCluster arguments>),RMSEs=list(<makeCluster arguments>)) allows full independent control of parallelisation for the two computations.

Value

refine returns an updated SLik or SLik_j object.

Note

See workflow examples in (by order of decreasing relevance) example_reftable, example_raw_proj and example_raw.

Examples

```
## see Note for links to examples.
```

---

rparam *Sample the parameter space*

Description

These functions take an SLik object (as produced by MSL) and samples its parameter space in (hopefully) clever ways, not yet well documented. rparam calls sample_volume to define points targeting the likelihood maximum and the bounds of confidence intervals, with n for these different targets dependent on the mean square error of prediction of likelihood at the maximum and at CI bounds.

Usage

```
rparam(object, n= 1, useEI = list(max=TRUE,profileCI=TRUE,rawCI=FALSE),
       useCI = TRUE, verbose = interactive(), tryn=30*n,
       level = 0.95, CIweight=Infusion.getOption("CIweight"))

sample_volume(object, n = 6, useEI, vertices=NULL,
              dlr = NULL, verbose = interactive(),
              fixed = NULL, tryn= 30*n)
```

Arguments

`object`	an `SLik` or `SLik_j` object
`n`	The number of parameter points to be produced
`useEI`	List of booleans, each determining whether to use an "expected improvement" (EI) criterion (e.g. Bingham et al., 2014) to select candidate parameter points to better ascertain a particular focal point. The elements `max`, `profileCI` and `rawCI` determine this for three types of focal points, respectively the MSL estimate, profile CI bounds, and full-dimensional bounds. When EI is used, `n` points with best EI are selected among `tryn` points randomly generated in some neighborhood of the focal point.
`vertices`	Points are sampled within a convex hull defined by `vertices`. By default, these vertices are taken from `object$fit$data`.
`useCI`	Whether to define points targeting the bounds of confidence intervals for the parameters. An expected improvement criterion is also used here.
`level`	If `useCI` is `TRUE` but confidence intervals are not available from the `object`, such intervals are computed with coverage `level`.
`dlr`	A (log)likelihood ratio threshold used to select points in the upper region of the likelihood surface. Default value is given by `Infusion.getOption("LRthreshold")`
`verbose`	Whether to display some information about selection of points, or not
`fixed`	A list or named vector, of which each element is of the form `<parameter name>=<value>`, defining a one-dimensional constraint in parameter space. Points will be sampled in the intersection of the volume defined by the `object` and of such constraint(s).
`tryn`	See `useEI` argument.
`CIweight`	For development purposes, not documented.

Value

a data frame of parameter points. Only parameters variable in the `SLik` object are considered.

References

D. Bingham, P. Ranjan, and W.J. Welch (2014) Design of Computer Experiments for Optimization, Estimation of Function Contours, and Related Objectives, pp. 109-124 in Statistics in Action: A Canadian Outlook (J.F. Lawless, ed.). Chapman and Hall/CRC.

Examples

```
if (Infusion.getOption("example_maxtime")>10) {
 data(densv)
 summliksurf <- infer_surface(densv) ## infer a log-likelihood surface
 sample_volume(summliksurf)
}
```

---

summLik *Model density evaluation for given data and parameters*

Description

Evaluation of inferred probability density as function of parameters and of (projected) summary statistics is implemented as a generic function `summLik`. Given the (projected) statistics for the data used to build the `SLik_j` object, and the fitted parameters, this returns the (log)likelihood as the generic `logLik` extractor. However, parameters can be varied (providing the likelihood function), and the data too.

This documentation deals mostly with the method for objects of class `SLik_j` produced by the up-to-date version of the summary-likelihood workflow.

Usage

```
summLik(object, parm, data, ...)

# S3 method for class 'SLik_j'
## S3 method for class 'SLik_j'
summLik(object, parm, data=t(attr(object$logLs,"stat.obs")),
                          log=TRUE, which="lik", ...)
```

Arguments

`object`	An `SLik` or `SLik_j` object
`parm`	data frame or matrix, containing coordinates of parameter points for which (log) likelihoods will be computed
`data`	The (projected, if relevant) summary statistics for which the likelihood of given parameters is to be computed. By default, the (projected) statistics for the data used to build the `SLik_j` object
`log`	Boolean: whether to return log likelihood or raw likelihood. Better ignored.
`which`	character string: `"lik"` for (log) likelihood inferred directly from the gaussian mixture model for joint parameters and summary statistics. But `"safe"`, which deals with a possible problem of this direct computation (see Details), is used internally by Infusion in all maximizations of likelihood.
`...`	further arguments passed to or from other methods.

Details

An object of class `SLik_j` contains a simulated joint distribution of parameters and (projected) summary statistics, and a fit of a multivariate gaussian mixture model to this simulated distribution, the "jointdens", from which a marginal density "margpardens" of parameters can be deduced. The raw likelihood(P;D) is the probability of the data D given the parameters P, viewed as function the parameters and for fixed data. It is inferred as jointdens(D,P)/margpardens(P) (for different P, each of jointdens and margdens are probabilities from a single (multivariate) gaussian mixture model, but this is not so for their ratio).

When margdens(P) is low, indicating that the region of parameter space around P has been poorly sampled in the simulation step, inference of likelihood is unreliable. Spuriously high likelihood may be inferred, which results notably in poor inference based on likelihood ratios. For this reason, it is often better to use the argument `which="safe"` whereby the likelihood is penalized when margdens(P) is low. The penalization is of the form
`penalized= unpenalized * pmin(1,margpardens/object$thr_dpar)`, where `thr_dpar` is a marginal density threshold stored in the `SLik_j` object. The source code should be consulted for details, and is subject to changes without notice.

Value

Numeric vector

See Also

`logLik`.

Examples

```
## Not run:
## Using 'slik_j' object from the example in help("example_reftable")
summLik(slik_j, parm=slik_j$MSL$MSLE+0.1)

# summLik() generalizes logLik():
summLik(slik_j, parm=slik_j$MSL$MSLE) == logLik(slik_j) # must be TRUE

## End(Not run)
```

`write_workflow` *Workflow template*

Description

codewrite_workflow writes a workflow script for inference. Beyond possibly saving some typing, this suggests what may be a reasonably good starting workflow. One should not expect to control all options of the workflow through the `write_workflow` arguments.

Usage

```
write_workflow(con = stdout(), lower, upper, nUnique, Simulate, simulator_args=NULL, ...)
```

Arguments

`con`	A connection object or a character string. Passed to `writeLines`.
`lower`	A named numeric vector of lower bounds for parameter space.
`upper`	A named numeric vector of upper bounds for parameter space.
`nUnique`	Number of simulations of the process (i.e. of rows of the reference table) in the first iteration.
`Simulate`	Function, or function name as a string. Sets the same-named `add_reftable` argument.
`simulator_args`	list of arguments for the simulator. Sets the `...` in the `add_reftable` call.
`...`	Sets additional arguments in the `refine` call.

Value

No return value. Used for the side-effect text, written to the connection.

Examples

```
write_workflow(
  ## arguments for init_reftable():
  lower=c(logTh1=-2,logTh2=-2,logTh3=-2,logTh4=-2,ar=0.01,logMu=-5,MEANP=0.01),
  upper=c(logTh1=1, logTh2=1, logTh3=1, logTh4=1, ar=0.99,logMu=-2,MEANP=0.99),
  nUnique = 1000,
  #
  ## for add_reftable():
  Simulate="schtroumf",   # name of a user-defined R function
 simulator_args= list(  # Imagine that schtroumf() require arguments 'exe_path' and 'cmdline':
    exe_path="'path_to_smurf_executable'",
    cmdline="'smurf.exe -a -b -c -d'"
  ),    # Do check the quotation marks in the output...
  #
  ## optional arguments for refine():
  n=8000/3.2, CIs=TRUE, update_projectors=FALSE)
```

Index