Package 'Infusion'

July 22, 2025

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, as first described in Rousset et al. (2017) <doi:10.1111/1755-0998.12627>. The package implements more advanced methods described in Rousset et al. (2025) <doi:10.1101/2024.09.30.615940>.

Encoding UTF-8

Version 2.3.0

Date 2025-07-22

Imports spaMM (>= 4.5.0), proxy, blackbox (>= 1.1.41), mvtnorm, methods, numDeriv, pbapply, ranger, foreach, matrixStats, boot, nloptr, geometry, cli, viridisLite

Suggests testthat, Rmixmod, caret, xLLiM, reticulate, mafR (>= 1.0.11)

Depends R (>= 3.3.0)

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

License CeCILL-2

ByteCompile true

URL https://gitlab.mbb.univ-montp2.fr/francois/Infusion

NeedsCompilation no

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

Repository CRAN

Date/Publication 2025-07-22 13:42:28 UTC

Contents

.update_obs *Updating an 'SLik_j' object for new data*

Description

.update_obs is an *experimental* utility to recycle the information in an inference object to produce an inference for different data without repeating a simulation workflow. Beyond its experimental nature, its result is not expected to provide the same precision of inference as a standard iterative **Infusion** workflow, since the recycled simulations were adapted to the original data only (.update_obs indeed allows one to investigate the effect of using this non-adapted information). So the new results should not be used as a equivalent to a full **Infusion** iterative workflow.

Usage

```
.update_obs(object, new.obs, CIs = FALSE, eval_RMSEs = FALSE, ...)
```

Arguments

object	an object of class "SLik_j" as createded by infer_SLik_joint (possible updated by MSL and refine).
new.obs	A named vector of summary statistics (ideally the projected ones, but raw ones are handled)
CIs, eval_RMSEs, ...	further arguments passed to MSL.

Value

An object of class "SLik_j" as the input object.

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, constr_crits=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 return a ***vector*** of summary statistics with named vector members, and it must have **one argument for each element** of the parameter vector (i.e. of each column of a matching `parsTable`).
`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.

constr_crits
: NULL, or quoted expression specifying a constraints on parameters, beyond the ones defined by the ranges over each parameter: see constr_crits for details. However, if sampled parameters were generated by init_reftable(., constr_crits=...), there is no need to apply the constraints again through add_reftable; and given the choice, it is better to apply them when calling init_reftable, as this allows a better control of the size of the reference table.

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'
              parsTable=parsp)

## Not run:  # example continued: parallel versions of the same.
# Slow cluster setup again
simuls <- add_simulation(NULL,Simulate="myrnorm_tab", parsTable=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

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

`allCIs` calls `confint` for all fitted parameters and re-structure the results.

Usage

```
allCIs(object, level=0.95, verbose=TRUE, ...)
## S3 method for class 'SLik_j'
confint(object, parm, level = 0.95, verbose = interactive(),
                        fixed = NULL, which = c(TRUE,TRUE), nsim = 0L,
                        reset = NULL, cluster_args = NULL, nb_cores = NULL,
                        type= if (nsim>1L) {"perc"} else NULL, ...)
## 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.
`nsim`	Integer: number of bootstrap replicates. If >1, bootstrap interval(s) are computed as further controlled by the `type` argument. Note that this will be ignored if the bootstrap has previously been run and `reset=FALSE`.
`reset`	Boolean: Whether to use any previously computed distribution (see Details) or not.
`cluster_args`, `nb_cores`	Passed to parallelization wrappers such as `dopar`.
`type`	Character vector: bootstrap CI type(s). Possible types are `"norm"`, `"basic"`, `"perc"` (as handled by `boot.ci`), and `"Bartlett"` (where the interval bounds are defined by threshold values of the likelihood ratio statistics modified using a Bartlett correction).
`...`	further arguments passed to or from other methods. `allCIs` passes them to `confint`, so that, e.g., `nsim` can be passed through the `....` `confint` passes them to parallelization wrappers such as `dopar`.

Details

`confint.SLik_j` results are stored in the `object` (until the next `refine`), including the result of the bootstrap simulation if it was performed. This distribution may then be reused by a next call to `confint` for the same `parm` if `reset=FALSE`. The default `reset` value is set such that the bootstrap distribution is recomputed only if `nsim` differs from the the stored number of simulations.

Bootstrap CIs computed using `boot.ci` are stored as distinct elements of the return list (see Value). However, for the `"Bartlett"` type of CI, the `interval` element of the return value is modified.

Value

Both functions modify the fit `object` as a side effect (see Details).

`confint` returns a list with sublists for each parameter, each sublist containing: the bounds of the one-dimensional confidence interval (element `interval`, a vector); the parameter point for the lower bound (element `lowerpar`, a vector including all parameters fitted in the `SLik` object), the full parameter point for the upper bound (element `upperpar`, formatted as `lowerpar`), and optionally if a bootstrap was run, the return value of a `boot::boot.ci` call (element `bootCI`) and the simulated distribution of parameter estimates (element `booreps`, 1-column matrix).

`allCIs` returns invisibly a list with elements including `CIs` (itself a list of `confint` results), `bounds` (a matrix made of bound points for all parameters), and some other elements.

See Also

SLRT

Examples

```
if (Infusion.getOption("example_maxtime")>3) {
#### Provide fit for minimal toy example:
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)
# simulated data with stands for the actual data to be analyzed:
ssize <- 40L
(Sobs <- myrnorm(mu=4,s2=1,sample.size=ssize) )
## Construct initial reference table:
# Uniform sampling in parameter space:
parsp <- init_reftable(lower=c(mu=2.8, s2=0.4, sample.size=ssize),
                       upper=c(mu=5.2, s2=2.4, sample.size=ssize))
# Build simulation table:
# set.seed(456)
simuls <- add_reftable(Simulate="myrnorm", parsTable=parsp)

# Infer surface:
densv <- infer_SLik_joint(simuls,stat.obs=Sobs)
# Usual workflow using inferred surface:
slik_1 <- MSL(densv) ## find the maximum of the log-likelihood surface

####  Confidence interval calculations:
(ci1 <- confint(slik_1,"mu")) # basic likelihood ratio interval
(ci2 <- confint(slik_1,"mu", nsim=199L)) # Percentile interval added
(ci3 <- confint(slik_1,"mu", nsim=199L, type="Bartlett")) # 'interval' corrected

# Previous bootstrap computations are stored in the fit object,
# and recycled if reset=FALSE *and* nsim > 0:
(ci4 <- confint(slik_1,"mu", nsim=199L, type= "Bartlett", reset=FALSE)) # = ci3
(ci5 <- confint(slik_1,"mu", nsim=199L, type= "perc", reset=FALSE)) # = ci2
}
```

constr_crits *Specificying arbitrary constraints on parameters*

Description

constr_crits is an argument of init_reftable, add_reftable, infer_SLik_joint, and profile.SLik_j, allowing to specify constraints on parameters, beyond the ones defined by the ranges over each parameter. Depending on the function it controls, this argument will affect the generation of parameter points or the maximization of the summary-likelihood.

It is advised to use the constr_crits when calling init_reftable so that the initial reference table satisfies the constraints. But it is more important to use it when calling infer_SLik_joint, whose return value contains the constr_crits information it was given, allowing subsequent refine calls to take constraints into account.

The constraints are represented as a mathematical expression for a vector of quantities that should all be negative when the constraints are satisfied. For example, to incorporate the constraint t1 < t3 && t2 < t3 between three time parameters t1, t2 and t3 present in the reference table, one can use constr_crits = quote({ c(t1-t3, t2-t3) }).

For computational efficiency, it may be better to avoid using this feature when the constraints can be represented as box constraints (i.e., independent ranges for each parameter) by some intuitive reparametrization.

Examples

```
## A toy example
if (Infusion.getOption("example_maxtime")>9) {
  myrnorm <- function(mu,s2,sample.size=40L) {
    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'

  # The constraint (the final plot produced by refine() explains it):
  heart <- quote({ x <- 3*(mu-4.25);  y <- 3*(s2-0.75); x^2+(y-(x^2)^(1/3))^2-1})

  set.seed(123)
  Sobs <- myrnorm(mu=4,s2=1)
  parsp_h <- init_reftable(lower=c(mu=2.8,s2=0.4), upper=c(mu=5.2,s2=2.4),
                           constr_crits=heart)
  simuls_h <- add_reftable(Simulate="myrnorm", parsTable=parsp_h)
  c_densv <- infer_SLik_joint(simuls_h, stat.obs=Sobs, constr_crits = heart)
  c_slik_j <- MSL(c_densv, CIs=FALSE, eval_RMSEs=FALSE)
  c_slik_j <- refine(c_slik_j, target_LR=10, ntot=3000, eval_RMSEs=FALSE)
}
```

`def_projectors` *Wrapper to generate projection functions for all parameters*

Description

Convenience function automating the initial `project.character` calls of a typical workflow.

Usage

```
def_projectors(reftable, pars, stats, npp = 1L, projNames = NULL,
               npp_opt=Infusion.getOption("npp_opt"), ...)
```

Arguments

`reftable`	A reference table of simulation as produced by `add_reftable`.
`pars`	Fitted parameter names for which projection functions should be returned.
`stats`	Names of summary statistics to be used for predicting parameter values.
`npp`, `npp_opt`	For development purposes.
`projNames`	Character vector of names of the projections. By default, names are given as `paste0("p", pars)` (or a more elaborate default when `npp > 1L`), and these names are checked for possible name conflicts with pre-existing variables in the `reftable`. Non-default values should be given in case of conflict.
`...`	Other arguments passed to `project.character`.

.

Details

The `npp > 1L` feature is experimental and awaiting further documentation.

Value

A list of *projectors*. Each of them is either an object returned by `project.character`, or a `list` inheriting from class `"projector"` which includes such a return object as one of its elements. The more detailed structure of a `"projector"` object is not part of the API.

Examples

```
## variation on example from help("example_reftable")

if (Infusion.getOption("example_maxtime")>2) {
  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=40)

  parsp_j <- init_reftable(lower=c(mu=2.5, s2=0.25, sample.size=40),
                           upper=c(mu=5.2, s2=2.4, sample.size=40))
  dsimuls <- add_reftable(,Simulate="blurred", parsTable=parsp_j,verbose=FALSE)

  ## Then instead of
  #
  # 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)
  #
  ## one can use
  #
  dprojectors <- def_projectors(dsimuls, pars=c("mu","s2"),
                   stats=c("mean","var"), npp = 1L,
                   projNames = c("MEAN", "VAR"))


  ## More speculatively, for non-default 'npp':
  dprojectors <- def_projectors(dsimuls, pars=c("mu","s2"),
                   stats=c("mean","var"), npp = 2L)
}
```

---

densv *Saved computations of inferred log-likelihoods*

Description

These are saved results from toy examples used in other documentation page for the package. They give 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'
#
## simulated data:
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", parsTable=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")>117) {
## 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", parsTable=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)

## 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`, to be preferred the primitive method described in the first publication of **Infusion**.

Examples

```
if (Infusion.getOption("example_maxtime")>56) {

  ## 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)))
  }

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

  ## Construct initial reference table:
  parsp_j <- init_reftable(lower=c(mu=2.5, s2=0.25, sample.size=40),
                           upper=c(mu=5.2, s2=2.4, sample.size=40))
  dsimuls <- add_reftable(,Simulate="blurred", parsTable=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)
plot_importance(slik_j, parm="mu")

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

## refine estimates iteratively
maxrefines <- 2L
# See get_workflow_design() for a suggested number of refine() calls,
# typically more than the 2 refine calls shown in this small example.
for(it in 1:maxrefines) slik_j <-
  refine(slik_j, eval_RMSEs=it==maxrefines, CIs=it==maxrefines)
##

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, but documented)
  predict(slik_j, newdata = slik_j$MSL$MSLE)   # the 'data' are here parameters!

  # 'ranger' projections can take a lot of memory. One can reduce them by...
  # deforest_projectors(slik_j)
  # ...before...
  # save(slik_j, file="slik_j")
```

```
    # They will be rebuilt on the fly if needed for further iterations.
  }
}
```

`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: identifier for the element to be extracted. See Examples for possible values.
`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

```
# ##### 0bserved summary statistics
# #  (raw data)
#   get_from(slik, "raw_data")
# #  (projected data, with raw ones as attribute, if relevant)
#   get_from(slik, which="obs") # or which="stat.obs" or "proj_data"
#
# ##### Reference-table information
# #  (raw = unprojected):
#   get_from(slik, "reftable_raw")
# # : but this is NULL if no projections were performed.
#
# # Projected:
#   get_from(slik, "reftable")
# # : including all parameters, latent variables, statistics,
# #   'cumul_iter' (the iteration in which each sample was added),
# #   and attributes.
#
# ##### RMSEs
# # 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.
#
# ##### Other, less commonly needed elements
#   get_from(slik, "Simulate"")
#   get_from(slik, "control.Simulate")
#   get_from(slik, "env")
#   get_from(slik, "packages")
# # => for these elements, the documentation and arguments of refine.default()
# #    provides meaning and context where they may be used.
#
# Number of elements of the multivariate gaussian mixture model:
#   get_from(slik, "nbCluster")
```

`get_LRboot` *Summary likelihood ratio tests*

Description

SLRT computes likelihood ratio tests based on the summary-likelihood surface and optionally on a fast bootstrap approximation implemented in get_LRboot (the latter function can be called directly but is rather typically called directly through SLRT). When bootstrapping is used, several correction of the basic likelihood ratio test may be reported, some more speculative than others, and bootstrap confidence intervals may be returned too.

get_LRboot provides a fast approximation to bootstrap distribution of likelihood ratio statistic (and optionally, of parameter estimates). 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.

Usage

```
SLRT(object, h0, nsim=0L, BGP=NULL, type="perc",
     level=0.95, nsteps=10L, variants=NULL, ...)
get_LRboot(object, h0_pars = NULL, nsim = 100L, reset = TRUE,
           BGP=object$MSL$MSLE, which="ecdf_2lr",
           bootCI.args=list(type="perc", conf = 0.95), ...)
```

Arguments

object	an SLik_j object.
h0	Numeric named vector of tested parameter values.
nsteps	Integer. Any value > 1 calls a profiling procedure (see Details).
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.
which	NULL or character string: controls the return value. If NULL, the function returns a list; otherwise, this specifies which element of the list to return.

type, level	vector of character strings: passed to boot.ci as arguments type and conf.
bootCI.args	list of arguments passed to boot.ci. Should not include the boot.out argument.
...	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).
variants	For development purposes, not documented.

Details

The computation of the likelihood ratio in high-dimensional models is easily confounded by maximization issues. Computation of a likelihood profile for a tested parameter between its summary-ML estimate and the tested value may be useful to reduce these issues. The number of steps of the profile is controlled by the nsteps value.

Bootstraps:

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 with default which argument 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. The chi2 statistic may bear as an attribute a solution vector value copied from the log-likelihood profile return value for the tested h0.

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;
bootCI	(present if h0 specified a single parameter and nsim>2) The result of boot::boot.ci, with slightly edited call element for conciseness.

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:
### *Old* 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", parsTable=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 `nbClu_pow_rule_fn` 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()` uses additional criteria involving the number of columns of the data to determine the maximum 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 `nbClu_pow_rule_fn` 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, nc=(nr/500+2)/3)
refine_nbCluster(nr, nc, onlymax=7)
get_nbCluster_range(projdata, nr = nrow(projdata), nc = ncol(projdata),
                    nbCluster = seq_nbCluster(nr, nc), verbose=TRUE)
```

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 (except if latent variables are included);
`nbCluster`	integer or vector of integers: candidate values, which feasability is checked by the function.
`verbose`	boolean. Whether to print some information, or not.

Details

The default upper value of the `nbCluster` range is controlled by two rules:

* The first rule sets the maximum number of clusters as function of the number of samples n in the reference table. The default rule `nr^(0.31-0.08/nc)` is close to the value $n^{0.3}$ irecommended in the `mixmod` statistical documentation (Mixmod Team, 2016).

* This first rule is corrected by a second rule setting a maximum dependent also on the dimensions of the `projdata` (the one used internally for clustering, which typically differs from the dimensions of the user-level `data`, if projections have been applied, in particular). This second rule is controlled by the `maxnbCluster` option.

For large number of points, experience shows that the maximum value derived from these two rules rules is practically always selected by AIC. So, in practice it is faster to only perform clustering with this maximum number of cluster, rather than to perform AIC-based selection among a range of number of clusters. This rule is implemented as the default for argument `nbCluster` of `refine.default`, by its default value specified by `refine_nbCluster`.

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
```

`get_workflow_design` *Workflow design*

Description

`get_workflow_design` provides default control values for the simulation plan, ideally chosen for good performance. The default design is ilustrated in the examples.

Usage

```
get_workflow_design(npar, n_proj_stats=npar, n_latent=0L,
                    final_reft_size=NULL,
                    refine_blocksize=NULL, subblock_nbr=NULL,
                    version=Infusion.getOption("version"),
                    cumn_over_maxit = NULL,
                    test_fac=NULL)
```

Arguments

`npar`	Number of fitted parameters of the statistical model.
`n_proj_stats`	number of projected summary statistics.
`n_latent`	Number of latent variables to be predicted.

`npar`, `n_proj_stats` and `n_latent` are here distinguished for clarity, but only their sum is currently used to define the sampling design.

`final_reft_size`	NULL, or integer specifying a non-default value of the final reference table size.
`refine_blocksize`	NULL, or integer specifying a non-default value of the number of points added per refine() call (except perhaps the first refine call in a workflow).
`subblock_nbr`	NULL, or integer specifying a non-default value of the target number of iterations per refine() call (actual number of iterations may differ).
`version`	A version number for **Infusion**. This is intended to allow reproducibility of results of various versions of **Infusion**.
`cumn_over_maxit`	logical; Whether to stop iteration when the target cumulative number of points is added to the reference table, or when the target number of iterations is first reached.
`test_fac`	optional numeric value; for testing a workflow, it may be useful to run it with smaller reference table sizes. `test_fac` specifies a reduction factor for these sizes, relative to the default design.

Value

`get_workflow_design` returns a list of control values, with elements `final_reft_size`, `init_reft_size`, `refine_blocksize`, `reftable_sizes`, and `subblock_nbr`, which can be used as shown in the Examples; as well as `cumn_over_maxit`, `first_refine_ntot`, and possibly other elements.

Examples

```
## This shows how the get_workflow_design() may be used,
## but in most cases one does not need to manipulate it.

## Not run:
  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)))
  }

  ## Simulated data:
  set.seed(123)
  dSobs <- blurred(mu=4,s2=1,sample.size=40)

  workflow_design <- get_workflow_design(npar=2L)

  ## Construct initial reference table:

  # Sample its parameters:
  if (IMPLICIT <- TRUE) { # use implicit control by get_workflow_design()
    parsp_j <- init_reftable(lower=c(mu=2.5,s2=0.25,sample.size=40),
                             upper=c(mu=5.2,s2=2.4,sample.size=40))
    # => get_workflow_design() has been called internally with default values
    #    to provide the dimension of the initial reference table.
    #    The following syntax provides a more explicit control:
  } else {
    parsp_j <- init_reftable(lower=c(mu=2.5,s2=0.25,sample.size=40),
                             upper=c(mu=5.2,s2=2.4,sample.size=4,
                             nUnique=workflow_design$init_reft_size))
  }
  # and yet another way to the same result could be
  #
  # parsp_j <- data.frame(mu=runif(init_reft_size,min=2.5,max=5.2),
  #                       s2=runif(init_reft_size,min=0.25,max=2.4),
  #                       sample.size=40)

  # Generate the initial simulations:
  dsimuls <- add_reftable(, Simulate="blurred", parsTable=parsp_j, verbose=FALSE)

  ## 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:
  # Initial Inference of log-likelihood surface
  slik_j <- infer_SLik_joint(dprojSimuls, stat.obs=dprojSobs, verbose=TRUE)
```

```
  # Find maximum, confidence intervals...
  slik_j <- MSL(slik_j, eval_RMSEs=FALSE, CIs=FALSE)

  ## Refinements over iterations
  # Here, with only two estimated parameters, workflow_design$final_reft_size
  # suggests a final reference table of 5000 simulations, attained through
  # 6 refine() calls with intermediate sizes given by
  (workflow_design$reftable_sizes)
  # here 500, 1000, 2000, 3000, 4000, 5000.
  #
  if (IMPLICIT) { # Again using implicit control by get_workflow_design()
    # Essentially, it suffices to call
    for (it in seq(6)) slik_j <-
      refine(slik_j, eval_RMSEs= it==6L, CIs= it==6L)
    # to run the default workflow. Again, the following syntax,
    # showing how successive table sizes are controlled internally,
    # provides a more explicit control:
  } else {
    reftable_sizes <- workflow_design$reftable_sizes
    init_reft_size <- workflow_design$init_reft_size
    refine_sizes <- diff(c(init_reft_size, reftable_sizes))
    maxit <- workflow_design$subblock_nbr
    for(it in seq_len(length(refine_sizes)-1L)) {
      add_it <- refine_sizes[it]
      slik_j <- refine(slik_j, ntot=add_it, maxit=maxit,
                       eval_RMSEs=FALSE, CIs=FALSE)
    }
    add_it <- tail(refine_sizes,1L)
    slik_j <- refine(slik_j, ntot=add_it, maxit=maxit,
                     CIs=add_it, eval_RMSEs=add_it)
  }

## End(Not run)
```

---

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.

`summary` and `print` methods for results of `goftest` call `str` to display the structure of this result.

Usage

```
goftest(object, nsim = 99L, method = "", stats=NULL, plot. = TRUE, nb_cores = NULL,
        Simulate = get_from(object,"Simulate"),
        control.Simulate=get_from(object,"control.Simulate"),
```

```
packages = get_from(object,"packages"),
env = get_from(object,"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 us 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.
`control.Simulate`	A list of arguments of the `Simulate` function (see`add_simulation`). The default value should generally be used, unless e.g. it contains the path of an executable on one machine and a different path must be specified on another machine.
`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 **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 **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 **s** under the fitted model, and of their projections **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 **u**, by estimating their joint distribution (using Gaussian mixture modelling) and letting t be the density of **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

An object of class `goftest`, which is a `list` with element(s)

`pval`	The p-value of the test (NULL if the test is not feasible).
`plotframe`	The data frame which is (by default) plotted by the function. Its last line contains the residuals u for the analyzed data, and other lines contain the bootstrap replicates.

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 section was written for the primitive workflow and may be largely irrelevant for the up-to-date one. It 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

The functions described here are either experimental or relevant only for the primitive workflow.

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 **spaMM** 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 Infusion.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), nc=ncol(data)) and mixmodGaussianModel=Inf 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 other forms of simulation-based inference such as Approximate Bayesian Computation. 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), nc=ncol(data)),
                 using = Infusion.getOption("mixturing"),
                 verbose = list(most=interactive(),pedantic=FALSE,final=FALSE),
                 marginalize = TRUE,
                 constr_crits=NULL,
                 projectors=NULL,
                 is_trainset)
```

Arguments

data	A data frame, whose each row contains a vector of parameters and one realization of the summary statistics for these parameters. Typically this holds the projected reference table (but see projectors argument for an experimental alternative).
stat.obs	Named numeric vector of observed values of summary statistics. Typically this holds the projected values (but see projectors argument for an experimental alternative).
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 (see add_reftable for more information on this function).
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", "EMCluster" or "mclust" to select the clustering methods used.
marginalize	Boolean; whether to derive the clustering of fitted parameters by marginalization of the joint clustering; if not, a distinct call to a clustering function is performed. It is strongly advised not to change the default. This argument might be deprecated in future versions.
constr_crits	NULL, or quoted expression specifying a constraints on parameters, beyond the ones defined by the ranges over each parameter: see constr_crits for details. This will control the parameter space both for maximization of the summary-likelihood, and for generation of new parameter points when refine() is called on the return object. See Examples section for a nice artificial toy example.
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.
projectors	if not NULL, this argument may be passed to project in the case the data or stat.obs do not yet contain the projected statistics. This **experimental** feature aims to remove the two user-level calls to project in the inference workflow.
is_trainset	Passed to project in the case the projectors argument is used.

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 reftable_raw of cumulated unprojected simulations.

See Also

constr_crits for using infer_SLik_joint in a model with non-box constraints on parameters.

Examples

```
if (Infusion.getOption("example_maxtime")>2) {
  myrnorm <- function(mu,s2,sample.size=40L) {
    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)
  # simulated data with stands for the actual data to be analyzed:
  Sobs <- myrnorm(mu=4,s2=1)
  # Uniform sampling in parameter space:
  parsp <- init_reftable(lower=c(mu=2.8,s2=0.4),
                         upper=c(mu=5.2,s2=2.4))
  # Build simulation table:
  simuls <- add_reftable(Simulate="myrnorm", parsTable=parsp)
  # Infer surface:
  densv <- infer_SLik_joint(simuls,stat.obs=Sobs)
  # Usual workflow using inferred surface:
 slik_j <- MSL(densv, eval_RMSEs=FALSE) ## find the maximum of the log-likelihood surface
  slik_j <- refine(slik_j, eval_RMSEs=FALSE)
  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

  # See also help for 'constr_crits'
}
```

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

Description

These functions are either for the primitive workflow or experimental. For standard use of **Infusion** see instead the functions used in the up-to-date workflow (`example_reftable`)

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 fitme (ρ, ν, ϕ, λ, and $etaFix=$$\beta$).
...	further arguments passed to or from other methods (currently not used).

Value

An object of class SLik or SLikp, which is a list including the fit object returned by fitme, 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.

The procedures for the "primitive" workflow, implemented in the first published version of the package, are being maintained for back compatibility, but users are urged to use the distinct, up-to-date workflow (see Examples). The package is expected to perform best when used in combination with **Rmixmod** for multivariate Gaussian mixture modeling, although the **mclust** package can also be used (with less control and perhaps decreased performance). The inference workflow typically (but not necessarily) includes dimension-reduction steps ("projections") for summary statistics, and has been more finely tuned for projections performed using the **ranger** package (though **Infusion** should handle other methods).

The up-to-date workflow builds and updates an object of class "SLik_j" which is designed to carry all the information required for pursuing the inference. Thus, successive inference steps (in particular, successive calls to the key refine function) can be carried on different computers, with the caveats noted below, by transferring the object between computers.

The "SLik_j" object includes in particular the reference table, information about all projections (this can be memory-expensive, so it has been made possible to remove non-essential information,

specifically for **ranger** results, using `deforest_projectors`: this information will be regenerated automatically if needed), and a sample-simulation function. If the latter function is a wrapper for an external simulation program, then this simulation program must be provided in addition to the `"SLik_j"` object. If an external program has to be called differently on different computers, information specific to each computer can be provided by the optional `control.Simulate` argument of `refine`.

People used to the functional-programming style common in R, where the return value entirely defines the effect of a function call, may be surprized by the distinct style of some of functions of this package. Indeed, the `"SLik_j"` objects include environments that may be modified by functions, independently of what these functions return. Notably, the profile plots (see `plot1Dprof`) and summary-likelihood ratio tests (`SLRT`) may update the summary-likelihood estimates. However, the more significant caveat of such usage of environments, from an inferential viewpoint, is that the projectors stored in input and output fit objects of a `refine` call are stored in the same environment, and therefore the projectors stored in the input object are modified in light of new simulations (or by any other operation affecting them in the refine), which alters the statistical meaning of any subsequent operation reusing the projectors from the input object. To keep the unmodified version of the projectors, users may need to save a fit object on disk before a refine call. This feature could of course be modified, but is retained for the following reasons: (1) in the routine workflow, only the latest fit object and its projectors will be relevant (but when comparing performance of the inference method for, e.g., reference tables of different sizes, more care is needed); and (2) projectors can be memory-expensive objects, so keeping distinct versions of them in successive fit objects may have substantial drawbacks.

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 more or less automatically handled by the up-to-date workflow, which also handles automatically NA values of summary statistics. For the primitive workflow, both of these problems could be handled to a limited extent using the `boundaries` attribute of the observed summary statistics (see `handling_NAs` for one 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. The current sampling strategy of these functions is crude but achieves desirable effects for present applications: it samples the space more uniformly than independent sampling of each point would, by generating fewer pairs of close points; and it is not exactly a regular grid.

`init_reftable` allows constraints to be applied on parameters. `init_grid` does not handle such constraints, and further generates replicates of `nRepl` of the parameter points sampled, which were required in the primitive workflow for good smoothing of the likelihood surface.

Usage

```
init_reftable(lower=c(par=0), upper=c(par=1), steps=NULL,
          nUnique=NULL,
          maxmin=(nUnique * length(lower)^2)<400000L,
          jitterFac=0.5, constr_crits=NULL, ...)
init_grid(lower=c(par=0), upper=c(par=1), steps=NULL, nUnique=NULL,
          nRepl=min(10L,nUnique), maxmin=TRUE, jitterFac=0.5)
```

Arguments

Arguments recommended for up-to-date workflow

`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`.
`nUnique`	Number of distinct values of parameter vectors in output. The default for `init_reftable` is determined by a call to `get_workflow_design` with `npar` argument being the number of variable parameters as determined by comparison of `lower` and `upper`. The result from `get_workflow_design` can be modified by passing further arguments to this function through the .... The default for `init_grid` is a less elaborate 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.
`constr_crits`	NULL, or quoted expression specifying a constraints on parameters, beyond the ones defined by the ranges over each parameter: see `constr_crits` for details.
`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.

Other, less useful arguments

`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.
`nRepl`	Number of replicates of distinct values of parameter vectors in output.
`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.
`...`	Further arguments passed to `get_workflow_design` when `nUnique` is left NULL. Can be used notably to specify non-default `n_proj_stats` or `n_latent`.

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_reftable(lower=c(mu=2.8,s2=0.5,sample.size=20),
              upper=c(mu=5.2,s2=4.5,sample.size=20))
# Less recommended:
init_reftable(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)
init_grid()
```

`latent` *Modeling and predicting latent variables*

Description

Latent variables are unobserved variables which, for given model parameters, are random. Since they are unobserved, they cannot appear in data nor used to infer parameters. However, they can be predicted if their joint distribution with the data is learned from the reference table. Thus, for inference about latent variables, these should be returned along summary statistics by the simulation function generating the samples for the reference table, but they should be declared as such so that later inference steps can distinguish them from both parameters and summary statistics. The `declare_latent` function is used for that purpose.

The pplatent function can be used to point-predict latent values, by their inferred mean or median given the (projected) summary statistics and fitted parameter values.

The latint function provides prediction intervals for the latent variable(s), accounting for uncertainty in parameter estimates, by using corrected levels obtained by a bootstrap method instead of the input levels (see Details). Sometimes the bounds cannot be determined from the simulations only, in particular when the corrected levels approach 0 or 1 (this may happen for example when the fitted distribution of the latent variable is degenerate with zero variance). In that case, NA will be returned instead of a quantile. If the user has additional knowledge about the distribution of the latent variable given the parameter estimates (e.g, if it has known finite bounds), s.he may use these corrected levels (included as attributes in the return value) to infer the interval bounds.

Usage

```
declare_latent(reftable, latentVars)
pplatent(object, type="mean",
         newDP=NULL,
         sumstats= t(get_from(object,"stat.obs")),
         pars=t(object$MSL$MSLE),
         size=1000L, ...)
latint(object, nsim=199L, levels=c(0.025,0.975),
       whichVars=object$colTypes$latentVars,
       sumstats= t(get_from(object,"stat.obs")),
       Simulate, control.Simulate=get_from(object,"control.Simulate"),
       bootSamples=NULL,
       ...)
```

Arguments

reftable	A reference table of simulation as returned by add_reftable
latentVars	A vector of names of variables to be treated as latent variables.
whichVars	A vector of names of latent variables for which intervals are to be computed.
object	An object of class SLik_j.
type	Character: the only handled non-default value is "median".
newDP, sumstats, pars	Matrices of data and/or (*projected*) summary statistics, with one column for each variable. newDP should contain both, and if NULL, is constructed from the next two arguments, sumstats holding statistics and pars holding parameters.
levels	Numeric vector: one-sided confidence levels (cumulative probabilities at which quantiles of the predictive distribution will be returned).
nsim	Integer: number of bootstrap replicates.
size	Integer: number of draws for estimating the median.
Simulate	May be used to provide the simulation function if it is not stored in the object; usage is then as for add_reftable. If it is set to NULL, bootstrap samples will instead be drawn from the gaussian mixture fit, but this shoudl be avoided for good performance. When the argument is missing (default), the Simulate information is sought in the object, and if absent, the gaussian mixture fit is used with a warning.

`control.Simulate`
: A list of arguments of the `Simulate` function (see `add_simulation`). The default value should be used unless you understand enough of its structure to modify it wisely (e.g., it may contain the path of an executable used to perform the fit on one machine and a different path may be specified to compute the prediction interval on another machine).

`bootSamples`
: A data frame. The bootstrap samples may be provided by this argument, otherwise they will be automatically simulated by the function provided by the `Simulate` argument. The boot samples may for example be obtained by calling `simulate(`*<Slik_j object>*`, SGP=TRUE)`.

`...`
: For `pplatent`: Not currently used. For `latint`: may be used to pass arguments `verbose`, `nb_cores`, `packages`, `env`, `control.Simulate`, `cluster_args`, and `cl_seed`, with usage as described for `add_reftable`, to the bootstrap sample simulation step. Parallelisation controls will also be used for the other steps of the bootstrap procedure.

Details

The `latint` function aims to provide intervals for the latent variable V, with better controlled coverage for given parameter values than would be provided by a plug-in method which would read the bounds of the intervals from the quantiles of the inferred distribution of latent values at the given confidence `levels`. For that purpose, corrected levels are used instead of the input `levels`.

Lawless & Fredette (2005) described a parametric bootstrap method to obtain these corrected levels, here adapted in the context of simulation-based inference. First, as in the original method, new data D^* and new latent values V^* are jointly simulated, given the summary-ML estimates (this uses the sample-generating process simulator, but these simulations are not added to the reference table). Second, their original method requires the evaluation for each new (D^*, V^*) of $F_V(V = V^*|D^*; \theta(D^*))$, the value of the cumulative distribution function F_V of V evaluated at $V = V^*$ values, given D^* and given new parameter estimates $\theta(D^*)$. It also requires the quantile function of V for the original data and parameter estimates. `latint` uses the fit of the multivariate gaussian mixture model to the reference table, stored in the fit object, for fast parameter refitting and for fast estimation of these functions on each bootstrap sample (D^*, V^*).

`latint(., levels=c(0.5))` will return the median of the predictive distribution, conceptually distinct from the plug-in prediction by `pplatent(slik_j, type="median")`

Quantile computations in `pplatent` and `latint` are approximate and might be modified in some future version, but were sufficient to obtain reasonable results in simulations.

Value

`declare_latent` returns the input `reftable` with modified attributes. `pplatent` returns a vector of predicted latent values.

`pplatent` returns a single numeric value or a vector. `latint` returns a list of matrices. Each matrix give quantiles of the predictive distribution, with one column per element of `levels`, one row per row of `sumstats`, and a `p_corr` attribute giving the vector of corrected levels used to obtain the quantiles (see Details). There is one such matrix for each latent variable.

References

Lawless J. F., Fredette M. (2005) Frequentist prediction intervals and predictive distributions. Biometrika 92: 529–542, <doi:10.1093/biomet/92.3.529>

Examples

```
## Not run:
  ##### A toy example motivated by some inference problem for genomic data #####
  # A model with three parameters logNe, Vs and Es is fitted.
  # Data from 100 loci are here summarized by three genome-wide summary statistics
  # (slogNe, sVs and sEs), and one locus-specific statistic that will provide
  # information about a locus-specific latent variable.

  ## Simulation function
  genomloc <- function(logNe=parvec["logNe"],Es=parvec["Es"],Vs=parvec["Vs"],
                       latent=TRUE, # returns the latent value by default
                       parvec) {
    slogNe <- rnorm(1,logNe, sd=0.3)
    genom_s <- rgamma(99, shape=Es/Vs,scale=Vs) # all loci except the focal one
    sEs <- mean(genom_s)
    sVs <- var(genom_s)
   latentv <- rgamma(1, shape=Es/Vs,scale=Vs) # locus-specific latent variable to predict
    sloc <- rnorm(1, mean=latentv-sEs,sd=latentv/3) # locus-specific statistic
    resu <- list(slogNe=slogNe,sEs=sEs,sVs=sVs, sloc=sloc)
    if (latent) resu$latentv <- latentv
    unlist(resu)
  }
  #
  ## simulated data, standing for the actual data to be analyzed:
  set.seed(123)
  Sobs <- genomloc(logNe=4,Es=0.05, Vs=0.1,latent=FALSE) ## no latent value
  #
  workflow_design <- get_workflow_design(npar=3L, n_proj_stats=4L, n_latent=1L)
  parsp <- init_reftable(lower=c(logNe=2,Es=0.001,Vs=0.001),
                         upper=c(logNe=6,Es=0.2,Vs=0.2),
                         nUnique=workflow_design$init_reft_size)
  simuls <- add_reftable(Simulate=genomloc, parsTable=parsp)

  simuls <- declare_latent(simuls,"latentv")

  ## Projections are not necessary here since the number of statistics is minimal,
  # but will be discussed later.
  { ############ Without projections
    { ## Usual workflow for estimation:
      densv <- infer_SLik_joint(simuls,stat.obs=Sobs)
      slik_j <- MSL(densv) ## find the maximum of the log-likelihood surface
      slik_j <- refine(slik_j,maxit=2,update_projectors=TRUE)
      # plot1Dprof(slik_j) ## 1D profiles show parameter inference is OK
    }
    { ## inference about latent values:
      pplatent(slik_j)
      pplatent(slik_j, type="median")
```

```
    latint(slik_j, nsim=999, levels=c(0.025,0.5,0.975))
  }
  { ## Assessing prediction of latent variable:
    # Builds testing set:
    test_simuls <- t(replicate(1000, genomloc(logNe=4,Es=0.05, Vs=0.1)))
    test_data <- test_simuls[,-5]
    # Point prediction:
    pred <- pplatent(slik_j, sumstats = test_data)

   plot(test_simuls[,"latentv"], pred); abline(0,1) # prediction vs. true latent values
  }
}

{ ########## Beyond standard use of projections for estimation of parameter values,
  # projections can also be used when several individual-level statistics inform about
  # the latent variable, to reduce them to a single summary statistic.
  # Projection will then be needed at the prediction step too.

  { # projection with latent variable as response:
    platent <- (project("latentv", data=simuls, stats=c("slogNe","sEs","sVs","sloc")))
    # (This example only serves to show the syntax since no dimention reduction occurs)

    dprojectors <- list(SLOC=platent,slogNe=NULL,sEs=NULL, sVs=NULL)

    # => As soon as one projection is used, The 'projectors' argument must include
  # all projectors used for the inference, whether for parameters or for latent variables.
    # NULL projectors should then be declared for raw statistics retained
    # in the projected reference table.

    # Apply projections on simulated statistics and 'data':
    projSimuls <- project(simuls,projectors=dprojectors,verbose=FALSE)
    projSobs <- project(Sobs,projectors=dprojectors)
  }

  { ## Estimation:
    ddensv <- infer_SLik_joint(projSimuls,stat.obs=projSobs)
    dslik_j <- MSL(ddensv) ## find the maximum of the log-likelihood surface
    dslik_j <- refine(dslik_j,maxit=2,update_projectors=TRUE)
    # plot1Dprof(dslik_j)
  }

  { ## Assessing prediction of latent variable: do not forget to project!

    test_simuls <- t(replicate(1000, genomloc(logNe=4,Es=0.05, Vs=0.1)))
    test_data <- test_simuls[,-5] # removing column of latent variable
    ptest_data <- project(test_data,projectors=dprojectors,verbose=FALSE) # Here!
    pred <- pplatent(dslik_j, sumstats = ptest_data)

    plot(test_simuls[,"latentv"], pred); abline(0,1)
  }
}
```

```
## End(Not run)
```

MAF.options *Control of MAF design and training*

Description

Masked Autoregressive Flows can be used by **Infusion** to infer various densities. This functionality requires the **mafR** package, and is requested through the `using` argument of `infer.SLik.joint` or `refine` (see Details).

`config_mafR` is a convenient way to reset the Python session (erasing all results stored in its memory), and in particular to enforce GPU usage.

`MAF.options` is a wrapper for `Infusion.options` which facilitates the specification of options that control the design of Masked Autoregressive Flows and their training.

Usage

```
config_mafR(torch_device, ...)
MAF.options(template = "zuko-like", ...)
```

Arguments

`torch_device`	character: `"cpu"`, `"cuda"` (or `"cuda:0"`, etc., useful if several GPUs are available) or `"mps"`.
`template`	Defines a template list of options, subsequently modified by options specified through the `...`, if any. Possible values of `template` are `"zuko-like"`, `"PSM19"`, `NULL`, or a `list` of options. The default value `"zuko-like"` reproduces **Infusion**'s defaults.
`...`	For `MAF.options`: a named value, or several of them, that override or complete the `template` values. For `config_mafR`: for development purposes; not documented.

Details

Possible `using` values (`using` being an argument of `infer.SLik.joint` or `refine`):

With `using="c.mafR"` four different MAFs are computed in every iteration: the density of statistics given parameters (providing the likelihood), the joint density, the instrumental density of parameters, and the "instrumental posterior" density of parameters given the data.

With `using="MAFmix"`, MAFs are computed only for the joint density and the instrumental density. The likelihood is deduced from them and a multivariate Gaussian mixture model is used to infer the "instrumental posterior" density.

`using="mafR"` can be used to let **Infusion** select one of the above options. `"MAFmix"` is currently called as it is faster, but this is liable to change in the future, so do not use this if you want a repeatable workflow.

Possible `template` values for `MAF.options`:

"PSM19" defines a template list of control values recommended by Papamakarios et al. (2019, Section 5.1). "zuko-like" defines a template list of values inspired from the tutorials of the **zuko** Python package, with some extra twists. Concretely, their definition is

```
if (template == "zuko-like") {
    optns <- list(design_hidden_layers = .design_hidden_layers_MGM_like,
       MAF_patience = 30, MAF_auto_layers = 3L, Adam_learning_rate = 0.001)
}
else if (template == "PSM19") {
    optns <- list(design_hidden_layers = .design_hidden_layers_PSM19,
       MAF_patience = 20, MAF_auto_layers = 5L, Adam_learning_rate = 1e-04)
}
```

and any replacement value should match the types (function, numeric, integer...) of the shown values, and the formals of the internal functions, in order to avoid cryptic errors.

The internal .design_hidden_layers... functions return a vector of numbers H_i of hidden values per layer i of the neural network. THevector has an attribute giving the resulting approximate number of parameters P of the deep-learning model according to Supplementary Table 1 of Papamakarios et al. 2017. H_i =50L for "PSM19". For "zuko-like", a typically higher value will be used. It is defined as a power of two such that P is of the order of 8 to 16 times the default number of parameters of the multivariate gaussian mixture model that could be used instead of MAFs.

Other controls which can be modified through the ... are
* MAF_validasize, a function which returns the size of the validation set, whose default definition returns 5% of its input value nr which is the number of samples in the reference table (consistently with Papamakarios et al., 2019);
* MAF_batchsize, a function that returns the batch size for the Adam optimizer. Its default simply returns 100L, but non-default functions can be defined, with at least the ... as formal arguments (more elaborate formals are possible but not part of the API).

Value

config_mafR is used for its side effects. Returns NULL invisibly.

MAF.options returns the result of calling Infusion.options on the arguments defined by the template and the .... Hence, it is a list of **previous** values of the affected options.

References

Papamakarios, G., T. Pavlakou, and I. Murray (2017) Masked Autoregressive Flow for Density Estimation. Pp. 2335–2344 in I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds. Advances in Neural Information Processing Systems 30. Curran Associates, Inc.
http://papers.nips.cc/paper/6828-masked-autoregressive-flow-for-density-estimation

Rozet, F., Divo, F., Schnake, S (2023) Zuko: Normalizing flows in PyTorch. https://doi.org/10.5281/zenodo.7625672

See Also

save_MAFs for saving and loading MAF objects.

Examples

```
  MAF.options(template = "zuko-like",
              Adam_learning_rate=1e-4,
              MAF_batchsize = function(...) 100L)

## Not run:
## These examples require the mafR package,
## and a Python installation with cuda capabilities.

if (requireNamespace("mafR", quietly=TRUE)) {

config_mafR() # set default: "cpu", i.e. GPU not used
config_mafR("cuda") # sets cuda as GPU backend
config_mafR() # reset python session, keeping latest backend


  config_mafR(torch_device="cuda")

  # function for sampling from N(.,sd=1)
  toyrnorm <- function(mu) {
    sam <- rnorm(n=40,mean=mu,sd=1)
    return(c(mean1=mean(sam)))
  }

  # simulated data, standing for the actual data to be analyzed:
  set.seed(123)
  Sobs <- toyrnorm(mu=4)

  parsp <- init_reftable(lower=c(mu=2.8),
                         upper=c(mu=5.2))
  simuls <- add_reftable(Simulate="toyrnorm", parsTable=parsp)

  MAF.options(template = "zuko-like")
  densv <- infer_SLik_joint(simuls,stat.obs=Sobs, using="mafR")

  # Usual workflow using inferred surface:
 slik_j <- MSL(densv, eval_RMSEs = FALSE) ## find the maximum of the log-likelihood surface
  # ETC.

  save_MAFs(slik_j, prefix = "toy_") # See its distinct documentation.
}

## End(Not run)
```

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 = prod(dim(object$logLs)) < 12000L, level = 0.95,
    verbose = interactive(),
    RMSE_n=Infusion.getOption("RMSE_nsim"),
    eval_RMSEs=(RMSE_n>1L) * prod(dim(object$logLs))<12000L,
    cluster_args=list(), nb_cores=NULL,
    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. See confint.SLik_j to obtain bootstrap confidence intervals.
level	Intended coverage probability of the confidence intervals.
verbose	Whether to display some information about progress and results.
RMSE_n	Integer: number of simulation replicates for evaluation of prediction uncertainty for likelihoods/ likelihood ratios/ parameters. The default value (10) provides quick but inaccurate estimates.
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).
nb_cores	Integer: shortcut for specifying cluster_args$spec.
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.

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 (possibly) 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 contains the `hessian` of summary likelihood at its maximum.

`CIobject` The return value of calling `allCIs`, converted to an environment.

`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 several of them, or a single unnamed argument which is a named list). The following values, with their defaults, are used in `Infusion`:

`version` A `package_version` object or something that can be converted to such an object by `as.package_version`. The default is the installed version of **Infusion**. Alternative values may be used by some functions (such as `get_workflow_design`) to use control values of the algorithms from other versions the the package. ;

`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 = 0.25`**:** 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_Ck"`**:** 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.

`global_strategy_args`**:** list of arguments for `Rmixmod::mixmodStrategy()`.

`nbClu_pow_rule_fn = function(nr) {...}` **and** `maxnbCluster = function(projdata, ...) {...}` functions controlling the value of `nbCluster` used in clustering by `Rmixmod` or other mixture modelling backends. See `get_nbCluster_range` for 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

`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 fit objects*

Description

Primarily 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_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,
                      show_latest=FALSE, ...)
# For objects returned by the primitive workflow:
## S3 method for class 'SLik'
plot(x, y, filled = FALSE, decorations = NULL,
                     color.palette = NULL, plot.axes = NULL,
                     plot.title = NULL, 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 required argument, giving the number of color levels. See Details for comments about default value.
`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)].
show_latest	Logical: whether to show distinctly the points added in the latest iteration.
...	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

The default color palette has been and may still be subject to changes, so if you want reproducible results, provide an explicit value. Some of the idiosyncratic color palettes used by (at least) early versions of **Infusion** helped me see what I want to see in a plot, but were not necessarily appropriate in all contexts. The current default for for "slice" plots in plot.SLik_j is function(n) grDevices::hcl.colors(n=n, palette="viridis", rev=TRUE). The "turbo" palette as called by function(n){ .viridisOpts(n=n, option="H", begin=0.1)} is being tried as default for filled, non-slice plots, and a more flashy one is used for dot plots.

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

plot.SLik_j returns invisibly a list including coordinates of the plot(s) (at least if the latest version of the spaMM package is installed). The exact format will depend on the nature of the plot but the names of elements should be self-explanatory. plot.SLik 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.

If you feel the 1D profiles are ugly, see the Details. Confidence intervals may still be correct in such cases.

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), notably when theyinclude many projections, and this may constrain the number of processes that can run in parallel.

Parallelization is also implemented for 1D profiles, but over the parameter for which profiles are computed, rather than over points in a profile. So it is effective only if profiles are computed for several parameters.

In default 2D plots, some areas may be left blank, for distinct reasons: the function values may be too low (as controlled by the `min_logLR` argument), or the likelihood profile may be maximized at parameter values which do not satisfy constraints defined by the `constr_crits` of the `infer_SLik_joint` function. Low profile values, even when shown, are not accurate anyway, since the inference workflow aims at inferring the top of likelihood "hill" relevant for the computation of confidence regions.

Usage

```
plot1Dprof(object, pars=object$colTypes$fittedPars, fixed=NULL,
           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,
           profiles=NULL, add=NULL, safe=TRUE,
           cluster_args=NULL, do_plot=TRUE, CIlevels=NULL,
           lower=NULL, upper=NULL,
           verbose=TRUE,
           ...)
plot2Dprof(object, pars=object$colTypes$fittedPars, fixed=NULL,
           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,
           min_logLR = qchisq(0.95,df=length(object$colTypes$fittedPars))/2 +3,
           lower=NULL, upper=NULL, color.palette=NULL, profiles=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: 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.
fixed	A named vector of parameter values to keep fixed in the profile computation: these parameters will be excluded from the pars as well as from the paramters over which maximization occurs.
type	Character: possible values are "logL" to plot the log-likelihood profile; "logLR" (or "LR" for the not-log version) to plot the log-likelihood-ratio profile; and, for 1D profiles, "zoom", "ranges" and "dual" which are variants of the "logLR" plot controlling the plot ranges in different ways (see Details).
gridSteps	The number of values (in each dimension for 2D plots) for 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 implementing graphic directives added to the plot (anything that is not a function is converted to the default function). Its formal parameters are as shown by its trivial default value, the par[.] arguments being parameters names (as a vector of character strings), to allow the additional plot elements to depend on the parameter of each subplot (see Plot 3 in Examples).
margefrac	For development purposes, not documented.
safe	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). For plot1Dprof, parallelisation is over the parametrs for which profiles are computed; For plot2Dprof,it is over the grid of points for each profile.
profiles	The profiles element of the return value of a previous call of the function. The point coordinates it provides for a given parameter or pair of parameters will be used, instead of recomputing the profile.
add	The profiles element of the return value of a previous plot1Dprof() call. The point coordinates it provides for a given parameter will be added to the profile produced by other arguments, using the graphic directives specified by the ...\ to distingush them.
min_logLR	Numeric; the minimal value of the log-likelihood ratio to be shown on 2D plots of type logLR, parameter regions having lower ratios being left blank.
do_plot	Boolean: whether to actually plot the profiles or only return computations for it.
CIlevels	NULL, or a vector of confidence levels whose bounds will be added to the plot. Suggested values are c(0.99, 0.95,0.90,0.75)
lower, upper	Numeric vectors of bounds for the parameters.
verbose	Boolean: whether to print information about the progress of the computation.
color.palette	Either NULL or a function that can replace the default color function used by plot2Dprof. The function must have a single argument, giving the number of color levels. See Details for comments about default value.
...	Further arguments passed by another function. Currently these arguments are ignored, except when handling the add argument, where they are passed to graphics::lines.

Details

Possible issues in computing the profiles: Computation of profiles is complicated by local maximization issues, which may result in highly visible artefacts in the plots. plot1Dprof tries to reduce their impact by computing the profile points starting from parameter values closest to the identified likelihood-maximizing value, and by using a method for selecting initial values for maximization which is partially controlled by the result of computation of the previous profile point. A second sequential computation of points profile may additionally be performed, this time starting from parameter values closest to the identified 95%\ confidence intervals rather than from the MLEs, when such intervals are available in the fit object or if they are requested by the CIlevels argument.

Graphic details of the plots: The 2D profile plots will typically include a contour level for the 95% 2D confidence region, labelled "2D CI" on the scale.

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 "ranges" (for plot1Dprof) is similar to "zoom" but maintains a fixed y range, facilitating comparison of profiles with and among different plot1Dprof calls. Few of the originally computed

points may appear in the retained x range, in which case it may be useful to generate additional points ensured to appear in the plot, using the `CIlevels` argument.

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`.

The default color palette has been and may still be subject to changes, so if you want reproducible results, provide an explicit value. The default palette for 2D profiles is currently the "turbo" palette as called by
`function(n){ .viridisOpts(n=n, option="H", begin=0.1)}`. If you want something more sequential,
`function(n) grDevices::hcl.colors(n=n, palette="viridis", rev=TRUE)` may be a better choice.

Value

Both functions return a list, invisibly for `plot1Dprof`. The list has elements
* `MSL_updated` which is a boolean indicating whether the summary-likelihood maximum has been recomputed (if it is TRUE, a message is printed);
* `profiles`, itself a list which stores information about each profile. The format of the information per profile is not yet stable (subject to changes without notice), but consistent with the one handled by the `profiles` and `add` argument). Its elements currently include
- the x,y or x,y,z coordinates of the putative plot;
- the full coordinates of the profile points in a `profpts` matrix (1D plot) or a 3D array (2D plot; first dimension are the fitted parameters, second and third are x and y steps).

`plot1Dprof` may have the side effect of adding confidence interval information to the fit object.

Examples

```
if (Infusion.getOption("example_maxtime")>20) { # 2D plots relatively slow
  #### 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)
  }
  #
  ## simulated data, 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))
  parsp <- cbind(parsp,sample.size=40)
  simuls <- add_reftable(Simulate="myrnorm2", parsTable=parsp)
```

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

 ### plots
 # Plot 1: a 1D profile:
prof1 <- plot1Dprof(slik_j,pars="s2",gridSteps=40,xlabs=list(s2=expression(paste(sigma^2))))

 # Using 'add' for comparison of successive profiles:
 slik_2 <- refine(slik_j, n=600)
 # Plot 2: comparing 1D profiles of different iterations:
prof2 <- plot1Dprof(slik_2,"s2", gridSteps=40,xlabs=list(s2=expression(paste(sigma^2))),
                    add=prof1$profiles, col="grey30")

 # Plot 3: using 'decorations' and 'profiles' for recycling previous computation for s2:
 DGpars <- c(mu1=4,mu2=2,s2=1,sample.size=40) # data-generating parameters
 plot1Dprof(slik_2, gridSteps=40,xlabs=list(s2=expression(paste(sigma^2))),
            profiles=prof2$profiles,
  decorations=function(par) {
     points(y=-7,x=DGpars[par],pch=20,cex=2,col="red");
     points(y=-0.5,x=slik_2$MSL$MSLE[par],pch=20,cex=2,col="blue")
  }
 )

 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")>5) {
 #### 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
prof1 <- plot1Dprof(slik,pars="s2",gridSteps=40,xlabs=list(s2=expression(paste(sigma^2))))
}
```

---

plot_proj *Diagnostic plots for projections*

Description

plot_proj and plot_importance are convenience functions providing diagnostic plots for projections, plot_proj plots predictions. This function is tailored for use on **ranger** projections, with out-of bag predictions for the training set shown in blue, and predictions for points outside the training set shown in black. Either parm or proj will then be needed to identify the projection to be plotted (perhaps both in some programming contexts, or if other projections methods are used).

plot_importance plots the importance metric variable.importance stored in a ranger object. Here, the projection may be identified by either parm or proj, or even directly provided as the object.

Usage

```
plot_proj(
  object, parm=NULL, proj,
  new_rawdata,
  use_oob=Infusion.getOption("use_oob"), is_trainset=FALSE,
  xlab=NULL, ylab=NULL, ...)

plot_importance(object, parm, proj, n.var = 30L, xlim=NULL,
                xlab = "Variable Importance", ylab = "", main="", ...)
```

Arguments

object	An object of class SLik_j. For plot_importance, it may also be an object of class ranger.
parm	Character string: a parameter name. Either one of parm or proj is required to get a diagnostic plot for a specific projector. Otherwise, a multipanel plot will be produced for all projectors.
proj	Character string: name of projected statistic.
new_rawdata	Reference table on which projections should be computed. If NULL, projections are not recomputed. Instead, the values stored in the object are used.
use_oob, is_trainset	Passed to project.default. Ignored if new_rawdata is NULL.
n.var	Integer: (maximum) number of predictor variables to be included in the plot.
xlim, xlab, ylab, main, ...	Passed to plot (for plot_proj) or to dotchart (for plot_importance).

Details

Projectors may be updated in an object after the projected statistics have been computed and included in the reference table, without any explicit action of the user on the object (see general information about **Infusion** for why and when this may occur). In particular the projectors stored in input and output fit objects of a refine call are stored in the same environment, and therefore the projectors stored in the input object are modified in light of new simulations. In that case, the plots produced by plot_proj may reflect properties either of the updated projections (if new_rawdata is used), or worse, may mix results of different projections (if new_rawdata is not used). A message or a warning may be issued when such events occur.

Value

These functions are mainly used for their side effect (the plot). `plot_importance` returns the vector of importance values invisibly. `plot_proj` returns invisibly, for each parameter a list with the `x`,`y` `xlab` and `ylab` elements of the `plot` call, or a structured list of such lists when plots are produced for several parameters.

Note

See workflow examples in `example_reftable`.

Examples

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

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

Description

As the Title says. As likelihood is obtained as a prediction from a statistical model for the likelihood surface, this has been 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, it has a `newdata` argument, as shared by many `predict` methods; but these `newdata` should be parameter values, not data). You can use the alternative `summLik` extractor if you do not like this syntax.

Usage

```
## S3 method for class 'SLik_j'
predict(
  object, newdata, log = TRUE, which = "safe",
  tstat = t(get_from(object,"stat.obs")),
  solve_t_chol_sigma_lists = object$clu_params$solve_t_chol_sigma_lists,
  constr_tuning= FALSE,
  ...)
```

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`	`"safe"` or `"lik"`. The default protects against some artefacts of extrapolation: see Details.

`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.

`constr_tuning` positive number, or FALSE: controls the effect of parameter constraints specified by the `constr_crits` argument of `infer_SLik_joint` on the evaluation of summary log-likelihood. When it is `FALSE` (or 0), no penalty is applied; when this is `Inf`, the log-likelihood of parameters violating constraints will be `-Inf`. Intermediate values allow an intermediate penalization (the source code should be consulted for further details), but their use is not recommended.

`...` For consistency with the generic. Currently ignored.

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 "pardens" 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)/pardens(P) (for different P, each of jointdens and pardens are probabilities from a single (multivariate) gaussian mixture model, but this is not so for their ratio).

When pardens(P) is low, indicating that the region of parameter space around P has been poorly sampled in the reference table, inference of likelihood in such regions 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 may be penalized where pardens(P) is low. The penalization is applied to cases where the uncorrected likelihood is higher than the maximum one in the reference table, and pardens(P) is lower than a threshold also determined from the reference table. The source code should be consulted for further details.

Value

Numeric: a single value, or a vector of (log-)likelihoods for different rows of the input `newdata`. When `which="safe"`, a `"lowdens"` attribute will be added when at least on parameter points had a low "pardens" (see Details).

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_j (or older SLik) object (as produced by MSL).

Usage

```
## S3 method for class 'SLik_j'
profile(fitted, value, fixed=NULL, return.optim=FALSE,
        init = "default", which="safe",
        constr_crits=fitted$constr_crits,
        eq_constr=NULL, ...)
## S3 method for class 'SLik'
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. Setting in explictly to NULL will maximize likelihood subject only to optional constraints specified by other arguments.
fixed	This argument appears redundant with the value argument, so it will be deprecated and its use should be avoided. When it is non-NULL, the profile is computed for value updated to c(value, fixed).
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.
constr_crits	Inequality constraints, by default those provided in the first iteration of the workflow, if any. See constr_crits for details.
eq_constr	Optional equality constraints, provided in the same format as constr_crits (**This feature is experimental**: in particular the procedure for finding initial values for maximization of likelihood does not use the eq_constr information).
...	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

If return.optim is FALSE (default): the predicted summary profile log-likelihood, with possible attribute "solution", the optimization solution vector (named numeric vector, missing if no profiling was needed). if return.optim is TRUE, the result of an optimization call, a list including elements solution (solution vector) and objective (log-likelihood).

See Also

example_reftable

Examples

```
## see e.g. 'example_reftable' documentation
```

---

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.

deforest_projectors is an utility to reduce the saved size of objects containing **ranger** objects (reproject can be used to reverse this).

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=eval(Infusion.getOption("proj_methodArgs")),
             verbose=TRUE, keep_data=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(), ext_projdata, ...)
get_projection(...) # alias for project.default

##
deforest_projectors(object)
```

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 parameter 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, whether 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, by setting is_trainset=TRUE.
is_trainset	Boolean. In a project call, set it to TRUE if x was used as the training set, to bypass a costly check (see use_oob argument). The same logic may apply in a plot_proj call, except that it is not immediately obvious for users whether the full reference table in an object was used as the training set, so trying to save time by setting is_trainset=TRUE requires more insight.
data	A data frame with all required variables, or, for the primitive workflow, a list of simulated empirical distributions, as produced by add_simulation.
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, so that if the data have more rows than trainingsize, 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. trainingsize is infinite by default for "ranger" method.
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 here, 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=.)). If "REML" or "GCV" methods are used, methodArgs is completely ignored. 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 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 two possible forms: (1) <name>=<project result>, where the <name> must differ from any name of x and <project result> is the return object of a project call; or (2) <name>=NULL where <name> is the name of a variable (raw summary statistic) in x (such explicit NULLs are **needed** for any raw statistic to be retained in the projected data; see Value).

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).

keep_data, ext_projdata

(experimental, and only when **ranger** is used). Setting keep_data=FALSE allows the input data to be removed from the return object of project.character (where they are otherwise part of its call element). This may be useful to save memory when multiple projections are based on the same data. However, as this data information is sometimes used, it must then be manually added as element projdata to the return value of infer_SLik_joint, and provided to project.default calls through the ext_projdata argument.

object An object of class SLik_j.

... Further arguments passed to or from other functions. Currently, 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 or no longer 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.

Predictions by 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 fitme 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 increases).

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, not checked for a loog time;
- `"keras"` deep learning using the `keras` package, not checked for a loog time.

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 the objective function of an `optim` call).

Value

`project.character` returns an object of the class returned by the called `method` (by default, a `ranger` object for the up-to-date workflow).

`project.default` returns an object of the same class and structure as the input `x`, containing the variables named in the `projectors` argument, each variable being a projected statistics inferred from the input summary statistics, or a summary statistic copied from the input `x` (if an **explicit** NULL projector was included for this statistic in the `projectors` argument).

`deforest_projectors` is used for its side effect (the contents of an environment within the input object been modifed), and returns a character string emphasizing this feature.

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.

See Also

plot_proj and plot_importance for diagnostic plots of the projections. def_projectors for automation of the project.character calls.

Examples

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

---

refine *Refine estimates iteratively*

Description

refine is a generic function with methods for objects of the classes produced by MSL. In the up-to-date workflow, it can automatically (1) define new parameters points, (2) add simulations to the reference table for these points, (3) optionally recompute projections, (4) update the inference of the likelihood surface, and (5) provides new point estimates, confidence intervals, and other results of an MSL call. It can repeat these steps iteratively as controlled by its workflow_design. Although it has many control arguments, few of them may be needed in any application. In particular it is designed to use reasonable default controls for the number of iterations, the number of points added in each iteration, and whether to update projections or not, when given only the current fit object as input.

reproject and recluster are wrappers for refine(..., ntot=0L), updating the object after either recomputing the projections or only re-performing the multivariate gaussian mixture clustering.

Usage

```
## S3 method for class 'SLik'
refine(object, method=NULL, ...)
## Default S3 method:
refine(
    object,
    ##       reference table simulations
    Simulate = attr(surfaceData,"Simulate"),
```

```
    control.Simulate = attr(surfaceData,"control.Simulate"),
    newsimuls = NULL,
    ##       CIs
    CIs = workflow_design$reftable_sizes[useCI],
    useCI = prod(dim(object$logLs))<12000L, level = 0.95,
    ##       workflow design
    workflow_design = get_workflow_design(
        npar=length(fittedPars), n_proj_stats = length(statNames),
        n_latent=length(latentVars)),
    maxit, ntot= maxit*.get_size_first_iter(object), n=NULL,
    ##       termination conditions
    precision = Infusion.getOption("precision"),
    eval_RMSEs = workflow_design$reftable_sizes,
    ##       verbosity
    verbose = list(most=interactive(),final=NULL, notable=TRUE, movie=FALSE,
                proj=FALSE, rparam=NULL, progress_bars=interactive()),
    ##       projection controls
    update_projectors = NULL,
    methodArgs = list(),
    ##       Likelihood surface modeling (up-to-date workflow)
    using = object$using,
    nbCluster = quote(refine_nbCluster(nr=nrow(data), nc=ncol(data))),
    ##       parallelisation
    cluster_args = list(), nb_cores=NULL, env=get_from(object,"env"),
    packages = get_from(object,"packages"), cl_seed=.update_seed(object),
    ##       obscure stuff
    target_LR = NULL,
    ##       not explicitly needed in up-to-date workflow
    trypoints = NULL,
    surfaceData,
    method,
    useEI = list(max=TRUE,profileCI=TRUE,rawCI=FALSE),
    rparamFn = Infusion.getOption("rparamFn"),
    ##
    ...
)

reproject(object, eval_RMSEs = NULL, CIs = NULL, ...)
recluster(object, eval_RMSEs = NULL, CIs = NULL, update_projectors=FALSE, ...)
```

Arguments

`object` an `SLik` or `SLik_j` object

reference table simulations

`Simulate` Character string: name of the function used to simulate samples. As it is typically stored in the object this argument does not need to be explicitly given; otherwise this should be the same function provided to `add_reftable`, whose documentation details the design requirements. 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.

control.Simulate

A list of arguments of the Simulate function (see add_simulation). The default value should be used unless you understand enough of its structure to modify it wisely (e.g., it may contain the path of an executable on one machine and a different path may be specified to refine a fit on another machine).

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 function extracted by get_from(object,"Simulate") 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.

CIs

CIs

Boolean, or boolean vector, or numeric (preferably integer) vector: controls to infer bounds of (one-dimensional, profile) confidence intervals. The numeric vector form allows to specify reference table size(s) for which CIs should be computed when these sizes are first reached. TRUE or FALSE will force or inhibit computation in all iterations. Finally (and probably less useful), a boolean vector such as CIs=c(TRUE,FALSE,TRUE) requests computation of CIs when the number of points cumulatively added reaches the target number of points for the first, third, and any subsequent iterations up to maxit (this may differ in certain cases from the first, third, and so on, iterations: see Details).

The default for refine is described in the Details. The default for reproject is to update the CIs if there are computed ones within the input object.

useCI

whether to perform RMSE computations for inferred confidence interval points.

level

Intended coverage of confidence intervals

workflow design

workflow_design

A list structured as the return value of get_workflow_design. The default value makes reference to elements of the input object's colTypes element.

maxit

Maximum number of iterative refinements (see also precision argument).

ntot

NULL or numeric: control of the total number of simulated samples (one for each new parameter point) to be added to the reference table over the maxit iterations. See Details for the rules used to determine the number of points added in each iteration. Reasonable default values are defined for ntot and maxit (see Details), so that beginners (and ideally, even more advanced users) do not have to find good values.

ntot=0L may be used to re-generate the projectors or the clustering without augmenting the reference table.

`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 in each iteration (see `n` argument of `sample_volume`). Slightly less intutive alternative to `ntot` specification, as there is at least one iteration where the actual number of added points is not the nominal `n` (see Details). `n=0L` will have the same effect as `ntot=0L`.

termination conditions

`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.

`eval_RMSEs` Same usage as for `CIs`; controls the `eval_RMSEs` argument of `MSL` in each iteration. See Details for the default. The default for `reproject` is to update the RMSEs if there are computed ones within the input `object`.

verbosity

`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`; `verbose$rparam` controls (cryptic) information about generation of new parameter points; `verbose$progress_bars` controls display of some progress bars. If `verbose` is an unnamed vector of booleans, they are interpreted as as-many first elements of the`verbose` vector, in the order shown by the default.

projection controls

`update_projectors`
Same usage as for `CIs`; this controls in which iterations the projectors are updated. The default `NULL` value is strongly recommended. See Details for further explanations.

`methodArgs` A list of arguments for the projection method. By default the `methodArgs` of the original `project.character` calls are reused over iteration, but elements of the new `methodArgs` list will be used to update the original `methodArgs`. Note that the updated list becomes the new default for further iterations.

Likelihood surface modeling

`using` Passed to `infer_SLik_joint`: a character string used to control the joint-density estimation method, as documented for that function (see `method` instead for equivalent control in primitive workflow). Default is to use to same method as in the the first iteration, but this argument allows a change of method.

nbCluster — Passed to infer_SLik_joint. The data in the expression for the default value refers to the data argument of the latter function.

parallelisation

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 see the methodArgs argument if independent control of parallelisation of the projection procedure is needed).

nb_cores — Integer: shortcut for specifying cluster_args$spec for sample simulation.

packages — NULL or a list with possible elements add_simulation and logL_method (the latter for the primitive workflow). These elements should be formatted as the packages arguments of add_simulation and infer_logLs, respectively, wherein they are the additional packages to be loaded on child processes. The effect of the default value of this argument is to pass over successive refine calls the value stored in the input fit object (itself determined by the latest use of the packages argument in, e.g., add_simulation or in previous refines).

env — An environment, passed as the env argument to add_simulation. The default value keeps the pre-refine value over iterations.

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.

others

target_LR — Likelihood ratio threshold used to control the sampling of new points and the selection of points for projections. Do not change it unless you known what you are doing.

method — For the primitive workflow: (a vector of) suggested method(s) for estimation of smoothing parameters (see method argument of infer_surface). 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.

trypoints — A data frame of parameters on which the simulation function get_from(object,"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.

useEI — for the primitive workflow only: cf this argument in rparam.

surfaceData — for the primitive workflow only: a data.frame with attributes, usually taken from the object and thus **not** specified by user, usable as input for infer_surface.

rparamFn — Function used to sample new parameter values.

... — further arguments passed to or from other methods. refine passes these arguments to the plot method suitable for the object.

Details

* **Controls of exploration of parameter space**: New parameter points are sampled so as to fill 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.

Each `refine` call performs several iterations, these iterations stopping when `ntot` points have been added to the simulation table. The target number of points potentially added in each iteration is controlled by the `ntot` and `maxit` arguments as described below, but fewer points may be actually added in each iteration, and more than `maxit` iterations may be needed to add the `ntot` points, if in a given iteration too few "good" candidate points are generated according to the internal rules for sampling the parameter region with high likelihood. In that case, the next iteration tries to keep up with the missing points by adding more points than the target number, but if not enough points have been added after `maxit` iterations, further iterations will be run.

CIs and RMSEs may be computed in any iteration but the default values of `eval_RMSEs` and `CIs` are chosen so as to avoid performing these computations too often, particularly when they are expected to be slow. The default implies that the RMSE for the maximum logL will be computed at the end each block of iterations that defines a refine (itself defined to reach to reference table sizes specified by the `workflow_design` and its default value). If the reference table is not too large (see default value of `useCI` for the precise condition), RMSEs of the logL are also computed at the inferred bounds of profile-based confidence intervals for each parameter.

Although the `update_projectors` argument allow similar control of the iterations where projections are updated, it is advised to keep it NULL (default value), so that whether projectors are updated in a given iteration is controlled by default internal rules. Setting it to `TRUE` would induce updating whenever any of the target reference table sizes implied by the `workflow_design$subblock_sizes` is reached. The default `NULL`, as the same effect subject to additional conditions: updating may not be performed when the training set is considered too similar to the one used to compute pre-existing projections, or when the train set includes more samples than the limit define by the global package option `upd_proj_subrows_thr`

Default values of `ntot` and `maxit` are controlled by the value of the `workflow_design`, which itself has the shown default value, and are distinct for the first vs. subsequent `refines`. The target number of points in each iteration is also controlled differently for the first vs. subsequent `refines`. This design is motivated by the fact that the likelihood surface is typically poorly inferred in the first refine so that the parameter points sampled then tend to be less relevant than those that can be sampled in later iterations. In the first `refine` call, the target number of points increases roughly as powers of two over iterations, to reach `ntot` cumulatively after `maxit` iterations. The default `ntot` is twice the size of the initial reference table, and the default `maxit` is 5. The `example_reftable` Example illustrates this, where the initial reference table holds 200 simulations, and the default target number of points to be added in 5 iterations by the first `refine` call are 25, 25, 50, 100 and 200. In later `refine` calls, the target number is `ntot/maxit` in each iteration.

* 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, distinct parallelisation 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). The most explicit way of specifying distinct controls is by a list structured as

```
cluster_args=list(reftable=list(<makeCluster arguments>),
```

RMSEs=list(<makeCluster arguments>))

A project=list(num.threads=<.>) element can be added to this list, providing control of the num.threads argument of **ranger** functions. However, this is retained mainly for back compatibility as the methodArgs argument can now be used to specify the num.threads.

Simpler arguments may be used and will be interpreted as follows: nb_cores, if given and not overriden by a spec argument in cluster_args (or in sublists of it), will control simulation and projection steps (but not RMSE computation): that is, nb_cores then gives the number of parallel processes for sample simulation, with additional makeCluster arguments taken from cluster_args, but RMSE computations are performed serially. On the other hand, a spec argument in cluster_args=list(spec=<.>, <other makeCluster arguments>)) will instead apply the same arguments to both reference table and RMSE computation, overcoming the default effect of nb_cores in both of them.

Value

refine returns an updated SLik or SLik_j object, unless both newsimuls and Simulate arguments are NULL, in which case a data frame of parameter points is returned.

Note

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

See get_workflow_design, the function that controls the default value of the workflow_design argument, and can be used to provide non-default controls.

Examples

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

---

reparam_fit *Conversion to new parameter spaces*

Description

Functions to facilitate inferences using alternative parametrizations of the same model, reusing an existing reference table. reparam_reftable produces a reference table in the new parametrization. reparam_fit does the same internally, and runs infer_SLik_joint and MSL on the new reference table.

reparam_fit is **experimental** and may have various limitations.

Usage

```
reparam_fit(fitobject, to, reparamfn,
            LOWER=NULL, UPPER=NULL, nbCluster="max",
            constr_crits = get_from(fitobject, "constr_crits"),
            raw=FALSE,
```

```
              reftable_attrs=NULL,
              ...)
reparam_reftable(fitobject, to, reparamfn,
              LOWER=NULL, UPPER=NULL, raw=FALSE, reftable_attrs=NULL)
```

Arguments

fitobject	an object of class SLik_j.
to	Character vector: names of all parameters in the new parametrization.
reparamfn	A function which can convert a data frame/matrix/vector of "old" parameters to an object of the same class in the new parametrization. It must have two arguments: its first argument must hold the "old" data frame/matrix/vector; and the second argument is either to (holding the to argument given to reparam_reftable) if reparamfn needs this information, or ....
LOWER, UPPER	Optional named vectors of bounds. They may be incomplete, containing values only for new parameters, and not necessarily for all of them (missing information is deduced from the observed ranges in the reference table).
nbCluster	Passed to infer_SLik_joint.
constr_crits	constr_crits applicable in the new parametrization. The suitability of such constraints is checked on the transformed reference table. When this argument is ignored, its default value is taken from the input object and therefore refers to the old parametrization. The check may then highlight the need for providing constraints redefined in reference to the new parametrization.
raw	Boolean; if TRUE, the object is re-built starting from the raw reference table. In particular the projections are re-computed.
reftable_attrs	A list whose elements are set as attributes to the re-parametrized reference table (see Value in add_reftable). Elements not provided by this argument will be copied from the input reference table. A typical use is to provide as Simulate function in the new parametrization (see Examples).
...	Passed to MSL.

Value

reparam_reftable returns a reference table with attributes, suitable as input for infer_SLik_joint. reparam_fit returns the return value of an MSL call.

The information about projections retained in these objects come from original fitobject.

Examples

```
## Not run:

## Toy simulation function
# (inspired by elementary population-genetic scenario)

hezsim <- function(logNe=parvec["logNe"],
                   logmu=parvec["logmu"],parvec) {
  Ne <- 10^logNe
```

```
  mu <- 10^logmu
  Es <- Ne*mu
  Vs <- 1/log(1+Ne)
  genom_s <- rgamma(5, shape=Es/Vs,scale=Vs) # 5 summary statistics
  names(genom_s) <- paste0("stat",seq(5))
  genom_s
}


{ ## Analysis with 'canonical' parameters
  #
  ## simulated data, standing for the actual data to be analyzed:
  set.seed(123)
  Sobs <- hezsim(logNe=4,logmu=-4)
  #
  parsp <- init_reftable(lower=c(logNe=1,logmu=-5),
                         upper=c(logNe=6,logmu=-2))
  init_reft_size <- nrow(parsp)
  simuls <- add_reftable(Simulate=hezsim, parsTable=parsp)

  {
    plogNe <- project("logNe", data=simuls, stats=paste0("stat",seq(5)))
    plogmu <- project("logmu", data=simuls, stats=paste0("stat",seq(5)))

    dprojectors <- list(plogNe=plogNe,plogmu=plogmu)

    projSimuls <- project(simuls,projectors=dprojectors,verbose=FALSE)
    projSobs <- project(Sobs,projectors=dprojectors)
  }

  { ## Estimation:
    ddensv <- infer_SLik_joint(projSimuls,stat.obs=projSobs)
   dslik_j <- MSL(ddensv, eval_RMSEs=FALSE) ## find the maximum of the log-likelihood surface
    refined_dslik_j <- refine(dslik_j, eval_RMSEs=FALSE, CIs=FALSE)
  }
}

{  ## Reparametrization to composite parameters

  locreparamfn <- function(object, ...) {
    logTh <- object[["logmu"]]+object[["logNe"]]
    if (inherits(object,"data.frame")) { # *data.frame case always needed.*
      data.frame(logTh=logTh,
                 logNe=object[["logNe"]])
    } else if (is.matrix(object)) {
      cbind(logTh=logTh,
            logNe=object[["logNe"]])
    } else c(logTh=logTh,
             logNe=object[["logNe"]])
  }

  { ## without re-projection
     rps <- reparam_fit(refined_dslik_j, to=c("logTh","logNe"),
```

```
                    reparamfn = locreparamfn)
   plot(rps)
}


{ ## with re-projection [necessary to allow refine()'s]

  # For refine() a new simulation will be needed, with new input parameters:
  hezsim2 <- function(logNe=parvec["logNe"],logTh=parvec["logTh"],parvec) {
    hezsim(logNe=logNe,logmu=logTh-logNe)
  }

  rps <- reparam_fit(refined_dslik_j, to=c("logTh","logNe"),
                     reparamfn = locreparamfn,
                     raw=TRUE,                  # to allow re-projection
                     reftable_attrs=list(Simulate=hezsim2))
  plot(rps)
  refine(rps)

}

}

## End(Not run)
```

---

rparam *Sample the parameter space*

Description

These functions are relevant only for the primitive workflow. They 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 objectfitdata.
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)
}
```

save_MAFs *Save or load MAF Python objects*

Description

Infusion fit results created using the **mafR** package contain (pointers to) Python objects, which are lost (their pointers being reduced to null pointers) when the fit object is saved on file and reloaded. The functions described here circumvent this issue.

`save_MAFs` will save the Python objects in distinct files. `load_MAFs` will load them back into the fit object, **not** overriding anyone pre-existing into the target fit object. The Python objects are saved under, and read from, files whose names are made of `"jointdens"`, `"pardens"`... and the given `prefix` and `extension`.

Usage

```
save_MAFs(object, ext="_MAF.pkl", prefix="")
load_MAFs(object, ext="_MAF.pkl", prefix="", set_path_only=FALSE)
```

Arguments

`object`	An object of class `"SLik_j"` as produced by the up-to-date **Infusion** workflow.
`prefix`, `ext`	Character: Prefix and extension for the filename of each saved MAF object.
`set_path_only`	Boolean: if TRUE, checks the presence of the saved MAF files, but do not load them, and (re)set the path information in the `object` (see Details for a programming context of usage).

Details

Both functions can write file path information into the input `object`'s `load_MAFs_info` element (an environment), where it can be read afterwards. `save_MAFs` will do so in all cases, and `load_MAFs` will do so when called with argument `set_path_only=TRUE`. When a bootstrap is run using parallelisation, the child processes can thus load files using their location information stored in this way. When the files have been moved after the `save_MAFs` call, their location information must then be updated using `load_MAFs(., set_path_onlTRUE)`.

Value

Both functions return the updated input `object`.

See Also

`reticulate::py_save_object` is used to save the Python objects.

Examples

```
## Given object 'slik_j' of class SLik_j
# save_MAFs(slik_j, prefix = "2024Feb30_")
## => '2024Feb30_jointdens_MAF.pkl', etc.
# save(slik_j, file="slik_j.rda")
## => Objects from Python environments are not saved in the RData file.

# and later:
# load(file="slik_j.rda")
# slik_j <- load_MAFs(slik_j, prefix = "2024Feb30_")
## Objects from pkl files are put back at the right place in 'slik_j'.
```

`simulate.SLik_j` *Simulate method for an* `SLik_j` *object.*

Description

`simulate` method for `SLik_j` objects, by default simulating realizations of the vector of projected summary statistics, drawn from their inferred distribution, given the summary-ML estimates which are the default value of the `given` argument.

For any non-default `given` argument, the sampling distribution is still deduced from the multivariate Gaussian mixture fit of the reference table, by conditioning it on `given` values. Any variable included in the mixture model may be included in `given`, allowing to simulate from other distributions than that of the vector of projected summary statistics.

This usage should not be confused with simulating the sample-generating process, necessarily distinctly available to the user, and which does not rely on the mixture model stored in the fit object. Simulations of the sample-generating process for `given` parameter values can be obtained by setting non-default option `SGP=TRUE`.

Usage

```
## S3 method for class 'SLik_j'
simulate(object, nsim = 1, seed = NULL, given=object$MSL$MSLE,
                           norm_or_t=.wrap_rmvnorm, SGP=FALSE, ...)
```

Arguments

`object`	An object of class `SLik_j` as produced by the up-to-date workflow.
`nsim`	number of response vectors of projected summary statistics to simulate.
`seed`	Seed for the random number generator (RNG). Here this controls the `.Random.seed` in the global environment, as in `simulate.lm`. This means that if a non-NULL `seed` is specified, it controls the RNG during the `simulate` call, but the RNG is reset to its prior state on exit.
`given`	The default is the summary-MLE, a full vector of fitted parameters; but Any variable included in the mixture fit of the referencetable may be included (see Description).

norm_or_t	Controls the sampler in in cluster of the mixture. The default value is a trivial wrapper around the rmvnorm sampler (consistently with the fitted model), but this argument makes it possible to specify other samplers (e.g., norm_or_t=Infusion:::.wrap_rmvt to sample from rmvt(., df=1); or used-defined samplers with the same interface).
SGP	Boolean. Whether to sample from the sample-generating process.
...	Additional arguments. Currently ignored, except when SGP=TRUE, in which case e.g. control.Simulate can be passed through the dots to control the sample simulator.

Value

By default (SGP=FALSE), a matrix of size nsim times the number of **projected** summary statistics; if SGP=TRUE, a data frame with columns for parameters, for **raw** summary statistics, and optionally for latent variables if relevant.

Examples

```
## Assuming an object 'slik_j' of class 'SLik_j':
# simulate(slik_j, nsim=3)
```

---

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

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 does. However, parameters can be varied (so that summLik provides the likelihood function rather than simply its maximum), the data can be varied too, and likelihood profiles (or even full new estimates) are computed when an incomplete parameter vector (or even NULL) is specified.

Usage

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

## S3 method for class 'SLik_j'
summLik(object, parm, data=t(get_from(object,"proj_data")),
                        log=TRUE, which="safe", constr_tuning = Inf,
                        newMSL=FALSE, ...)
```

Arguments

object	An SLik or SLik_j object
parm	Vector, data frame or matrix, containing coordinates of parameter points for which (log) likelihoods will be computed; or NULL. A profile will be computed if a single incomplete parameter vector is provided. A full new estimate will be computed in parm is NULL.
data	Matrix of (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 deduced from the multivariate gaussian mixture model for joint parameters and summary statistics, without further modifications. But the default, "safe", may correct this result to deal with possible extrapolation artefacts (see Details of predict.SLik_j).
constr_tuning	Passed to predict.SLik_j.
newMSL	Boolean. If this is TRUE and a profile was computed, attributes are added to the result (see Value).
...	further arguments passed to or from other methods. Currently only passed to predict.SLik_j when no likelihood profile is computed.

Value

Numeric vector, with optional attribute(s).

If no profile is computed, it may have attributes from the return value of predict.SLik_j. If a profile is computed, the returned value has attribute "profpt" giving the profile-maximizing parameter vector. Further, if newMSL=TRUE, the following attributes are added: "newobs_MSL", a list with information about unconstrained summary-likelihood maximization (useful mainly when there are new data); and "LRstat", the resulting log-likelihood ratio.

See Also

predict.SLik_j) for case without profiling; logLik, the standard extractor of likelihood for the model fitted to the original data.

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)
```

Index