Package 'mixComp'

November 18, 2020

Version 0.1-1

Title Estimation of the Order of Mixture Distributions

Description Methods for estimating the order of a mixture model. The approaches considered are based on the following papers (the extended list of references is available in the vignette):
1. Dacunha-Castelle, Didier, and Elisabeth Gassiat. The estimation of the order of a mixture model. Bernoulli 3, no. 3 (1997): 279-299. <https://projecteuclid.org/download/pdf_1/euclid.bj/1177334456>.
2. Woo, Mi-Ja, and T. N. Sriram. Robust estimation of mixture complexity. Journal of the American Statistical Association 101, no. 476 (2006): 1475-1486. <doi:10.1198/016214506000000555>.
3. Woo, Mi-Ja, and T. N. Sriram. Robust estimation of mixture complexity for count data. Computational statistics & data analysis 51, no. 9 (2007): 4379-4392. <doi:10.1016/j.csda.2006.06.006>.
4. Umashanger, T., and T. N. Sriram. L2E estimation of mixture complexity for count data. Computational statistics & data analysis 53, no. 12 (2009): 4243-4254. <doi:10.1016/j.csda.2009.05.013>.
5. Karlis, Dimitris, and Evdokia Xekalaki. On testing for the number of components in a mixed Poisson model. Annals of the Institute of Statistical Mathematics 51, no. 1 (1999): 149-162. <doi:10.1023/A:1003839420071>.
6. Cutler, Adele, and Olga I. Cordero-Brana. Minimum Hellinger Distance Estimation for Finite Mixture Models. Journal of the American Statistical Association 91, no. 436 (1996): 1716-1723. <doi:10.2307/2291601>.

Imports cluster, boot, expm, matrixcalc, Rsolnp, kdensity

Suggests knitr, rmarkdown

License GPL-3

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

NeedsCompilation no

Author Anja Weigel [aut],
Yulia Kulagina [aut, cre],
Fadoua Balabdaoui [aut, ths],
Lilian Mueller [ctb],

Martin Maechler [ctb] (package 'nor1mix' as model,
<https://orcid.org/0000-0002-8685-9910>)

Maintainer Yulia Kulagina <yulia.kulagina@stat.math.ethz.ch>

Repository CRAN

Date/Publication 2020-11-18 10:00:02 UTC

R topics documented:

`datMix` *Create Object for Which to Estimate the Mixture Complexity*

Description

Function to generate a `datMix` object to be passed to other `mixComp` functions used for estimating the mixture complexity.

Usage

```
datMix(dat, dist, theta.bound.list = NULL, MLE.function = NULL, Hankel.method = NULL,
                                                                Hankel.function = NULL)

is.datMix(x)

## S3 method for class 'datMix'
print(x, ...)
```

Arguments

`dat`	a numeric vector containing the observations from the mixture model.

`dist` a character string giving the (abbreviated) name of the component distribution, such that the function `ddist` evaluates its density function and `rdist` generates random numbers. For example, to create a gaussian mixture, `dist` has to be specified as `norm` instead of `normal`, `gaussian` etc. for the package to find the functions `dnorm` and `rnorm`.

`theta.bound.list` a named list specifying the upper and the lower bound for the component parameters. The names of the list elements have to match the names of the formal arguments of the functions `ddist` and `rdist` exactly. For a gaussian mixture, the list elements would have to be named `mean` and `sd`, as these are the formal arguments used by `rnorm` and `dnorm`. Has to be supplied if a method that estimates the component weights and parameters is to be used.

`MLE.function` function (or list of functions) which takes as input the data and gives as output the maximum likelihood estimator for the parameter(s) of a one component mixture (i.e. the standard MLE of the component distribution `dist`). If the component distribution has more than one parameter, a list of functions has to be supplied and the order of the MLE functions has to match the order of the component parameters in `theta.bound.list` (e.g. for a normal mixture, if the first entry of `theta.bound.list` is the bounds of the mean, then then first entry of `MLE.function` has to be the MLE of the mean). If this argument is supplied and the `datMix` object is handed over to a complexity estimation procedure relying on optimizing over a likelihood function, the `MLE.function` attribute will be used for the single component case. In case the objective function is either not a likelihood or corresponds to a mixture with more than 1 components, numerical optimization will be used based on `Rsolnp`'s function `solnp`, but `MLE.function` will be used to calculate the initial values passed to `solnp`. Specifying `MLE.function` is optional and if it is not, for example because the MLE solution does not exists in closed form, numerical optimization is used to find the relevant MLE'.

`Hankel.method` character string in `c("explicit","translation","scale")`, specifying the method of estimating the moments of the mixing distribution used to calculate the relevant Hankel matrix. Has to be specified when using `nonparamHankel`, `paramHankel` or `paramHankel.scaled`. For further details see below.

`Hankel.function` function needed for the moment estimation via `Hankel.method`. This normally depends on `Hankel.method` as well as `dist`. For further details see below.

`x` **in** `is.datMix()`**:** R object.
in `print.datMix()`**:** object of class `datMix`.

`...` further arguments passed to the print method.

Details

If the `datMix` object is supposed to be passed to a function that calculates the Hankel matrix of the moments of the mixing distribution (i.e. `nonparamHankel`, `paramHankel` or `paramHankel.scaled`), the arguments `Hankel.method` and `Hankel.function` have to be specified. The `Hankel.methods` that can be used to generate the estimate of the (raw) moments of the mixing distribution and the corresponding `Hankel.functions` are the following, where j specifies an estimate of the number of components:

`"explicit"` For this method, `Hankel.function` contains a function with arguments called `dat` and `j`, explicitly estimating the moments of the mixing distribution from the data and the currently assumed mixture complexity. Note that what Dacunha-Castelle & Gassiat (1997) called the "natural" estimator in their original paper is equivalent to using `"explicit"` with `Hankel.function`

$$f_j((1/n) * sum_i(\psi_j(X_i))).$$

`"translation"` This method corresponds to Dacunha-Castelle & Gassiat's (1997) example 3.1. It is applicable if the family of component distributions (G_θ) is given by

$$dG_\theta(x) = dG(x - \theta),$$

where G is a known probability distribution whose moments can be given explicitly. `Hankel.function` contains a function of j returning the jth (raw) moment of G.

`"scale"` This method corresponds to Dacunha-Castelle & Gassiat's (1997) example 3.2. It is applicable if the family of component distributions (G_θ) is given by

$$dG_\theta(x) = dG(x/\theta),$$

where G is a known probability distribution whose moments can be given explicitly. `Hankel.function` contains a function of j returning the jth (raw) moment of G.

If the `datMix` object is supposed to be passed to a function that estimates the component weights and parameters (i.e. all but `nonparamHankel`), the argument `theta.bound.list` has to be specified, and `MLE.function` will be used in the estimation process if it is supplied (otherwise the MLE is found numerically). Note that the `datMix` function will change the random number generator (RNG) state.

Value

An object of class `datMix` with the following attributes (for further explanations see above):

`dist`

`discrete` logical indicating whether the underlying mixture distribution is discrete.

`theta.bound.list`

`MLE.function`

`Hankel.method`

`Hankel.function`

See Also

`RtoDat` for the conversion of `rMix` to `datMix` objects.

Examples

```
## observations from a (presumed) mixture model
obs <- faithful$waiting
```

```
## generate list of parameter bounds (assuming gaussian components)
norm.bound.list <- vector(mode = "list", length = 2)
names(norm.bound.list) <- c("mean", "sd")
norm.bound.list$mean <- c(-Inf, Inf)
norm.bound.list$sd <- c(0, Inf)
## generate MLE functions
# for "mean"
MLE.norm.mean <- function(dat) mean(dat)
# for "sd" (the sd function uses (n-1) as denominator)
MLE.norm.sd <- function(dat){
  sqrt((length(dat) - 1) / length(dat)) * sd(dat)
}
# combining the functions to a list
MLE.norm.list <- list("MLE.norm.mean" = MLE.norm.mean,
                      "MLE.norm.sd" = MLE.norm.sd)

## function giving the j^th raw moment of the standard normal distribution,
## needed for calculation of the Hankel matrix via the "translation" method
## (assuming gaussian components with variance 1)

mom.std.norm <- function(j){
  ifelse(j %% 2 == 0, prod(seq(1, j - 1, by = 2)), 0)
}

## generate 'datMix' object
faithful.dM <- datMix(obs, dist = "norm", theta.bound.list = norm.bound.list,
                      MLE.function = MLE.norm.list, Hankel.method = "translation",
                      Hankel.function = mom.std.norm)

## using 'datMix' object to estimate the mixture complexity
set.seed(1)
res <- paramHankel.scaled(faithful.dM)
plot(res)
```

---

dMix *Mixture density*

Description

Evaluate the (log) density function of a mixture specified as `Mix` object.

Usage

```
dMix(x, obj, log = FALSE)
```

Arguments

`x`	vector of quantiles.
`obj`	object of class `Mix`.
`log`	logical; if `TRUE`, probabilities/densities f are returned as $log(f)$.

Value

dMix(x) returns the numeric vector of probability values $f(x)$, logged if log is TRUE.

See Also

Mix for the construction of Mix objects, rMix for random number generation (and construction of an rMix object) and plot.Mix which makes use of dMix.

Examples

```
# define 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

# evaluate density at points x
x <- seq(7, 20, length = 501)
dens <- dMix(x, normLocMix)
plot(x, dens, type = "l")

# compare to plot.Mix
plot(normLocMix)
```

---

hellinger.cont *Estimate a Continuous Mixture's Complexity Based on Hellinger Distance*

Description

Estimation of a continuous mixture's complexity as well as its component weights and parameters by minimizing the squared Hellinger distance to a kernel density estimate.

Usage

```
hellinger.cont(obj, bandwidth, j.max = 10, threshold = "SBC", sample.n = 5000,
               sample.plot = FALSE, control = c(trace = 0))

hellinger.boot.cont(obj, bandwidth, j.max = 10, B = 100, ql = 0.025,
                    qu = 0.975, sample.n = 3000, sample.plot = FALSE,
                    control = c(trace = 0), ...)
```

Arguments

obj	object of class datMix.
bandwidth	numeric indicating the bandwidth to be used. Can also be set to "adaptive" if the adaptive kernel density estimator as defined by Cutler & Cordero-Brana (1996, page 1720, Equation 2) should be employed.
j.max	integer stating the maximal number of components to be considered.

`threshold`	function or character string in `c("AIC","SBC")` specifying which threshold should be used to compare two mixture estimates of complexities j and $j+1$. If the difference in minimized squared distances is smaller than the relevant threshold, j will be returned as complexity estimate.
`sample.n`	integer specifying the sample size to be used for approximation of the objective function (see details).
`sample.plot`	logical indicating whether the histogram of the sample drawn to approximate the objective function should be plotted.
`control`	control list of optimization parameters, see `solnp`.
`B`	integer specifying the number of bootstrap replicates.
`ql`	numeric between 0 and 1 specifying the lower quantile to which the observed difference in minimized squared distances will be compared.
`qu`	numeric between 0 and 1 specifying the upper quantile to which the observed difference in minimized squared distances will be compared.
`...`	further arguments passed to the `boot` function.

Details

Define the *complexity* of a finite continuous mixture F as the smallest integer p, such that its probability density function (pdf) f can be written as

$$f(x) = w_1 * g(x;\theta_1) + \ldots + w_p * g(x;\theta_p).$$

Further, let g, f be two probability density functions. The squared Hellinger distance between g and f is given by

$$H^2(g,f) = \int (\sqrt{g(x)} - \sqrt{f(x)})^2 = 2 - 2\int \sqrt{f(x)}\sqrt{g(x)},$$

where $\sqrt{g(x)}$, respectively $\sqrt{f(x)}$ denotes the square root of the probability density functions at point x. To estimate p, `hellinger.cont` iteratively increases the assumed complexity j and finds the "best" estimate for both, the pdf of a mixture with j and $j+1$ components, ideally by calculating the parameters that minimize the squared Hellinger distances to a kernel density estimate. Since the computational burden of optimizing over an integral to find the "best" component weights and parameters is immense, the algorithm approximates the objective function by sampling `sample.n` observations Y_i from the kernel density estimate and using

$$2 - 2\sum \sqrt{f(Y_i)}/\sqrt{g(Y_i)},$$

instead, with f being the mixture density and g being the kernel density estimate. Once the "best" parameters have been obtained, the difference in squared distances is compared to a predefined `threshold`. If this difference is smaller than the threshold, the algorithm terminates and the true complexity is estimated as j, otherwise j is increased by 1 and the procedure is started over. The predefined thresholds are the `"AIC"` given by

$$(d+1)/n$$

and the `"SBC"` given by

$$((d+1)log(n))/(2n),$$

n being the sample size and d the number of component parameters, i.e. θ is in R^d. Note that, if a customized function is to be used, it may only take the arguments `j` and `n`, so if the user wants to include the number of component parameters d, it has to be entered explicitly. `hellinger.boot.cont` works similarly to `hellinger.cont` with the exception that the difference in squared distances is not compared to a predefined threshold but a value generated by a bootstrap procedure. At every iteration (of j), the function sequentially tests $p = j$ versus $p = j+1$ for $j = 1, 2, \ldots$, using a parametric bootstrap to generate `B` samples of size n from a j-component mixture given the previously calculated "best" parameter values. For each of the bootstrap samples, again the "best" estimates corresponding to pdfs with j and $j + 1$ components are calculated, as well as their difference in approximated squared Hellinger distances from the kernel density estimate. The null hypothesis $H_0 : p = j$ is rejected and j increased by 1 if the original difference in squared distances lies outside of the interval $[ql, qu]$, specified by the `ql` and `qu` empirical quantiles of the bootstrapped differences. Otherwise, j is returned as the complexity estimate. To calculate the minimum of the Hellinger distance (and the corresponding parameter values), the solver `solnp` is used. The initial values supplied to the solver are calculated as follows: the data is clustered into j groups by the function `clara` and the data corresponding to each group is given to `MLE.function` (if supplied to the `datMix` object `obj`, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights and the MLE's are taken as initial component parameter estimates.

Value

Object of class `paramEst` with the following attributes:

`dat`	data based on which the complexity is estimated.
`dist`	character string stating the (abbreviated) name of the component distribution, such that the function `ddist` evaluates its density function and `rdist` generates random numbers.
`ndistparams`	integer specifying the number of parameters identifying the component distribution, i.e. if θ is in R^d then `ndistparams`$= d$.
`formals.dist`	string vector specifying the names of the formal arguments identifying the distribution `dist` and used in `ddist` and `rdist`, e.g. for a gaussian mixture (`dist = norm`) amounts to `mean` and `sd`, as these are the formal arguments used by `dnorm` and `rnorm`.
`discrete`	logical indicating whether the underlying mixture distribution is discrete. Will always be `FALSE` in this case.
`mle.fct`	attribute `MLE.function` of `obj`.
`pars`	Say the complexity estimate is equal to some j. Then `pars` is a numeric vector of size $(d+1)*j-1$ specifying the component weight and parameter estimates, given as $(w_1, \ldots w_{j-1}, \theta 1_1, \ldots \theta 1_j, \theta 2_1, \ldots \theta d_j)$.
`values`	numeric vector of function values gone through during optimization at iteration j, the last entry being the value at the optimum.
`convergence`	indicates whether the solver has converged (0) or not (1 or 2) at iteration j.

References

Details can be found in

1. M.-J. Woo and T. Sriram, "Robust Estimation of Mixture Complexity", Journal of the American Statistical Association, Vol. 101, No. 476, 1475-1486, Dec. 2006.
2. A. Cutler, O.I. Cordero-Brana, "Minimum Hellinger Distance Estimation for Finite Mixture Models." Journal of the American Statistical Association, Vol. 91, No. 436, 1716-1723, Dec. 1996.

See Also

`hellinger.disc` for the same estimation method for discrete mixtures, `solnp` for the solver, `datMix` for the creation of the `datMix` object.

Examples

```
### generating 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

### generating 'rMix' from 'Mix' object (with 1000 observations)
set.seed(1)
normLocRMix <- rMix(10000, normLocMix)

### generating 'datMix' from 'R' object

## generate list of parameter bounds

norm.bound.list <- vector(mode = "list", length = 2)
names(norm.bound.list) <- c("mean", "sd")
norm.bound.list$mean <- c(-Inf, Inf)
norm.bound.list$sd <- c(0, Inf)

## generate MLE functions

# for "mean"
MLE.norm.mean <- function(dat) mean(dat)
# for "sd" (the sd function uses (n-1) as denominator)
MLE.norm.sd <- function(dat){
sqrt((length(dat) - 1) / length(dat)) * sd(dat)
}
# combining the functions to a list
MLE.norm.list <- list("MLE.norm.mean" = MLE.norm.mean,
                      "MLE.norm.sd" = MLE.norm.sd)

## generating 'datMix' object
normLoc.dM <- RtoDat(normLocRMix, theta.bound.list = norm.bound.list,
                     MLE.function = MLE.norm.list)


### complexity and parameter estimation
```

```
## Not run:
set.seed(0)
res <- hellinger.cont(normLoc.dM, bandwidth = 0.5, sample.n = 5000)
plot(res)

## End(Not run)
```

`hellinger.disc` *Estimate a Discrete Mixture's Complexity Based on Hellinger Distance*

Description

Estimation of a discrete mixture's complexity as well as its component weights and parameters by minimizing the squared Hellinger distance to the empirical probability mass function.

Usage

```
hellinger.disc(obj, j.max = 10, threshold = "SBC",
                                  control = c(trace = 0))

hellinger.boot.disc(obj, j.max = 10, B = 100, ql = 0.025, qu = 0.975,
                                        control = c(trace = 0), ...)
```

Arguments

`obj`	object of class `datMix`.
`j.max`	integer stating the maximal number of components to be considered.
`threshold`	function or character string in `c("AIC","SBC")` specifying which threshold should be used to compare two mixture estimates of complexities j and $j+1$. If the difference in minimized squared distances is smaller than the relevant threshold, j will be returned as complexity estimate.
`control`	control list of optimization parameters, see `solnp`.
`B`	integer specifying the number of bootstrap replicates.
`ql`	numeric between 0 and 1 specifying the lower quantile to which the observed difference in minimized squared distances will be compared.
`qu`	numeric between 0 and 1 specifying the upper quantile to which the observed difference in minimized squared distances will be compared.
`...`	further arguments passed to the `boot` function.

Details

Define the *complexity* of a finite discrete mixture F as the smallest integer p, such that its probability mass function (pmf) f can be written as

$$f(x) = w_1 * g(x; \theta_1) + \ldots + w_p * g(x; \theta_p).$$

Further, let g, f be two probability mass functions. The squared Hellinger distance between g and f is given by

$$H^2(g, f) = \sum (\sqrt{g(x)} - \sqrt{f(x)})^2,$$

where $\sqrt{g(x)}$, respectively $\sqrt{f(x)}$ denotes the square root of the probability mass functions at point x. To estimate p, `hellinger.disc` iteratively increases the assumed complexity j and finds the "best" estimate for both, the pmf of a mixture with j and $j + 1$ components, by calculating the parameters that minimize the squared Hellinger distances to the empirical probability mass function. Once these parameters have been obtained, the difference in squared distances is compared to a predefined `threshold`. If this difference is smaller than the threshold, the algorithm terminates and the true complexity is estimated as j, otherwise j is increased by 1 and the procedure is started over. The predefined thresholds are the `"AIC"` given by

$$(d + 1)/n$$

and the `"SBC"` given by

$$((d + 1)log(n))/(2n),$$

n being the sample size and d the number of component parameters, i.e. θ is in R^d. Note that, if a customized function is to be used, it may only take the arguments `j` and `n`, so if the user wants to include the number of component parameters d, it has to be entered explicitly. `hellinger.boot.disc` works similarly to `hellinger.disc` with the exception that the difference in squared distances is not compared to a predefined threshold but a value generated by a bootstrap procedure. At every iteration (of j), the function sequentially tests $p = j$ versus $p = j + 1$ for $j = 1, 2, \ldots$, using a parametric bootstrap to generate `B` samples of size n from a j-component mixture given the previously calculated "best" parameter values. For each of the bootstrap samples, again the "best" estimates corresponding to pmfs with j and $j + 1$ components are calculated, as well as their difference in squared Hellinger distances from the empirical probability mass function. The null hypothesis $H_0 : p = j$ is rejected and j increased by 1 if the original difference in squared distances lies outside of the interval $[ql, qu]$, specified by the `ql` and `qu` empirical quantiles of the bootstrapped differences. Otherwise, j is returned as the complexity estimate. To calculate the minimum of the Hellinger distance (and the corresponding parameter values), the solver `solnp` is used. The initial values supplied to the solver are calculated as follows: the data is clustered into j groups by the function `clara` and the data corresponding to each group is given to `MLE.function` (if supplied to the `datMix` object `obj`, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights and the MLE's are taken as initial component parameter estimates.

Value

Object of class `paramEst` with the following attributes:

`dat` data based on which the complexity is estimated.

dist	character string stating the (abbreviated) name of the component distribution, such that the function ddist evaluates its density function and rdist generates random numbers.
ndistparams	integer specifying the number of parameters identifying the component distribution, i.e. if θ is in R^d then ndistparams$= d$.
formals.dist	string vector specifying the names of the formal arguments identifying the distribution dist and used in ddist and rdist, e.g. for a gaussian mixture (dist = norm) amounts to mean and sd, as these are the formal arguments used by dnorm and rnorm.
discrete	logical indicating whether the underlying mixture distribution is discrete. Will always be TRUE in this case.
mle.fct	attribute MLE.function of obj.
pars	Say the complexity estimate is equal to some j. Then pars is a numeric vector of size $(d+1)*j-1$ specifying the component weight and parameter estimates, given as $$(w_1, ...w_{j-1}, \theta 1_1, ...\theta 1_j, \theta 2_1, ...\theta d_j).$$
values	numeric vector of function values gone through during optimization at iteration j, the last entry being the value at the optimum.
convergence	indicates whether the solver has converged (0) or not (1 or 2) at iteration j.

References

M.-J. Woo and T. Sriram, "Robust estimation of mixture complexity for count data", Computational Statistics and Data Analysis 51, 4379-4392, 2007.

See Also

L2.disc for the same estimation method using the L2 distance, hellinger.cont for the same estimation method for continuous mixtures, solnp for the solver, datMix for the creation of the datMix object.

Examples

```
## create 'Mix' object
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

## create random data based on 'Mix' object (gives back 'rMix' object)
set.seed(0)
poisRMix <- rMix(1000, obj = poisMix)

## create 'datMix' object for estimation

# generate list of parameter bounds
poisList <- vector(mode = "list", length = 1)
names(poisList) <- "lambda"
poisList$lambda <- c(0, Inf)
```

```
# generate MLE function
MLE.pois <- function(dat){
  mean(dat)
}

# generating 'datMix' object
pois.dM <- RtoDat(poisRMix, theta.bound.list = poisList, MLE.function = MLE.pois)

## complexity and parameter estimation
set.seed(0)
res <- hellinger.disc(pois.dM)
plot(res)
```

`L2.disc` *Estimate a Discrete Mixture's Complexity Based on L2 Distance*

Description

Estimation of a discrete mixture's complexity as well as its component weights and parameters by minimizing the squared L2 distance to the empirical probability mass function.

Usage

```
L2.disc(obj, j.max = 10, n.inf = 1000, threshold = "SBC", control = c(trace = 0))

L2.boot.disc(obj, j.max = 10, n.inf = 1000, B = 100, ql = 0.025, qu = 0.975,
                                                  control = c(trace = 0), ...)
```

Arguments

`obj`	object of class `datMix`.
`j.max`	integer stating the maximal number of components to be considered.
`n.inf`	integer; the L2 distance contains an infinite sum, which will be approximated by a sum ranging from 0 to `n.inf`.
`threshold`	function or character string in `c("LIC","SBC")` specifying which threshold should be used to compare two mixture estimates of complexities j and $j+1$. If the difference in minimized squared distances is smaller than the relevant threshold, j will be returned as complexity estimate.
`control`	control list of optimization parameters, see `solnp`.
`B`	integer specifying the number of bootstrap replicates.
`ql`	numeric between 0 and 1 specifying the lower quantile to which the observed difference in minimized squared distances will be compared.
`qu`	numeric between 0 and 1 specifying the upper quantile to which the observed difference in minimized squared distances will be compared.
`...`	further arguments passed to the `boot` function.

Details

Define the *complexity* of a finite discrete mixture F as the smallest integer p, such that its probability mass function (pmf) f can be written as

$$f(x) = w_1 * g(x; \theta_1) + \ldots + w_p * g(x; \theta_p).$$

Further, let g, f be two probability mass functions. The squared L2 distance between g and f is given by

$$L_2^2(g, f) = \sum (g(x) - f(x))^2.$$

To estimate p, `L2.disc` iteratively increases the assumed complexity j and finds the "best" estimate for both, the pmf of a mixture with j and $j + 1$ components, by calculating the parameters that minimize the squared L2 distances to the empirical probability mass function. The infinite sum contained in the objective function will be approximated by a sum ranging from 0 to `n.inf`, set to 1000 by default. Once the "best" parameters have been obtained, the difference in squared distances is compared to a predefined `threshold`. If this difference is smaller than the threshold, the algorithm terminates and the true complexity is estimated as j, otherwise j is increased by 1 and the procedure is started over. The predefined thresholds are the `"LIC"` given by

$$(0.6 * log((j + 1)/j))/n$$

and the `"SBC"` given by

$$(0.6 * log(n) * log((j + 1)/j))/n,$$

n being the sample size. Note that, if a customized function is to be used, it may only take the arguments `j` and `n`. `L2.boot.disc` works similarly to `L2.disc` with the exception that the difference in squared distances is not compared to a predefined threshold but a value generated by a bootstrap procedure. At every iteration (of j), the function sequentially tests $p = j$ versus $p = j + 1$ for $j = 1, 2, \ldots$, using a parametric bootstrap to generate `B` samples of size n from a j-component mixture given the previously calculated "best" parameter values. For each of the bootstrap samples, again the "best" estimates corresponding to pmfs with j and $j + 1$ components are calculated, as well as their difference in squared L2 distances from the empirical probability mass function. The null hypothesis $H_0 : p = j$ is rejected and j increased by 1 if the original difference in squared distances lies outside of the interval $[ql, qu]$, specified by the `ql` and `qu` empirical quantiles of the bootstrapped differences. Otherwise, j is returned as the complexity estimate. To calculate the minimum of the L2 distance (and the corresponding parameter values), the solver `solnp` is used. The initial values supplied to the solver are calculated as follows: the data is clustered into j groups by the function `clara` and the data corresponding to each group is given to `MLE.function` (if supplied to the `datMix` object `obj`, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights and the MLE's are taken as initial component parameter estimates.

Value

Object of class `paramEst` with the following attributes:

`dat`	data based on which the complexity is estimated.
`dist`	character string stating the (abbreviated) name of the component distribution, such that the function `ddist` evaluates its density function and `rdist` generates random numbers.

ndistparams	integer specifying the number of parameters identifying the component distribution, i.e. if θ is in R^d then ndistparams$= d$.
formals.dist	string vector specifying the names of the formal arguments identifying the distribution dist and used in ddist and rdist, e.g. for a gaussian mixture (dist = norm) amounts to mean and sd, as these are the formal arguments used by dnorm and rnorm.
discrete	logical indicating whether the underlying mixture distribution is discrete. Will always be TRUE in this case.
mle.fct	attribute MLE.function of obj.
pars	Say the complexity estimate is equal to some j. Then pars is a numeric vector of size $(d+1)*j-1$ specifying the component weight and parameter estimates, given as $$(w_1, ...w_{j-1}, \theta 1_1, ...\theta 1_j, \theta 2_1, ...\theta d_j).$$
values	numeric vector of function values gone through during optimization at iteration j, the last entry being the value at the optimum.
convergence	indicates whether the solver has converged (0) or not (1 or 2) at iteration j.

References

T. Umashanger and T. Sriram, "L2E estimation of mixture complexity for count data", Computational Statistics and Data Analysis 51, 4379-4392, 2007.

See Also

hellinger.disc for the same estimation method using the Hellinger distance, solnp for the solver, datMix for the creation of the datMix object.

Examples

```
## create 'Mix' object
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

## create random data based on 'Mix' object (gives back 'rMix' object)
set.seed(1)
poisRMix <- rMix(10000, obj = poisMix)

## create 'datMix' object for estimation
# generate list of parameter bounds
poisList <- vector(mode = "list", length = 1)
names(poisList) <- "lambda"
poisList$lambda <- c(0, Inf)

# generate MLE function
MLE.pois <- function(dat){
  mean(dat)
}

# generating 'datMix' object
```

```
pois.dM <- RtoDat(poisRMix, theta.bound.list = poisList, MLE.function = MLE.pois)


## complexity and parameter estimation

set.seed(1)
res <- L2.disc(pois.dM)
plot(res)
```

Mix *Mixtures of Univariate Distributions*

Description

Objects of class `Mix` represent finite mixtures of any univariate distribution. Methods for construction, printing and plotting are provided.

`plot` method for `Mix` objects visualizing the mixture density, optionally showing the component densities.

Usage

```
Mix(dist, w=NULL, theta.list=NULL, name=NULL, ...)

is.Mix(x)

## S3 method for class 'Mix'
plot(
  x,
  ylim,
  xlim = NULL,
  xout = NULL,
  n = 511,
  type = NULL,
  xlab = "x",
  ylab = "f(x)",
  main = attr(obj, "name"),
  lwd = 1.4,
  log = FALSE,
  components = TRUE,
  h0 = FALSE,
  parComp = list(col = NULL, lty = 3, lwd = 1),
  parH0 = list(col = NULL, lty = 3, lwd = 1),
  ...
)

## S3 method for class 'Mix'
print(x, ...)
```

Arguments

dist	a character string giving the (abbreviated) name of the component distribution, such that the function ddist evaluates its density function and rdist generates random numbers. For example, to create a gaussian mixture, dist has to be specified as norm instead of normal, gaussian etc. for the package to find the functions dnorm and rnorm.
w	numeric vector of length p, specifying the mixture weights $w[i]$ of the components, $i = 1, \ldots, p$. If the weights don't add up to 1, they will be scaled accordingly. Uses equal weights for all components by default.
theta.list	named list specifying the component parameters. The names of the list elements have to match the names of the formal arguments of the functions ddist and rdist exactly. For a gaussian mixture, the list elements would have to be named mean and sd, as these are the formal arguments used by rnorm and dnorm. Alternatively, the component parameters can be supplied directly as named vectors of length p via ...
name	optional name tag of the result (used for printing and plotting).
...	further arguments passed to the function plotting the mixture density.
x	object of class Mix.
ylim	range of y values to use; if not specified (or containing NA), the function tries to construct reasonable default values.
xlim	range of x values to use; particularly important if xout is not specified. If both are left unspecified, the function tries to construct reasonable default values.
xout	numeric or NULL giving the abscissae at which to draw the density.
n	number of points to generate if xout is unspecified (for continuous distributions).
type	character denoting the type of plot, see e.g. lines. Defaults to "l" if the mixture distribution is continuous and to "h" otherwise.
xlab, ylab	labels for the x and y axis with defaults.
main	main title of plot, defaulting to the Mix name.
lwd	line width for plotting with a non-standard default.
log	logical; if TRUE, probabilities/densities f are plotted as $log(f)$. Only works if components is set to FALSE.
components	logical indicating whether the individual mixture components should be plotted, set to TRUE by default.
h0	logical indicating whether the line $y = 0$ should be drawn.
parComp	graphical parameters for drawing the individual components if components is set to TRUE.
parH0	graphical parameters for drawing the line $y = 0$ if h0 is set to TRUE.

Details

Note that the Mix function will change the random number generator (RNG) state.

Value

An object of class `Mix` (implemented as a matrix) with the following attributes:

`dim`	dimensions of the matrix.
`dimnames`	a `dimnames` attribute for the matrix.
`name`	as entered via `name`.
`dist`	as entered via `dist`.
`discrete`	logical indicating whether the mixture distribution is discrete.
`theta.list`	as entered via `theta.list`.

See Also

`dMix` for the density, `rMix` for random numbers (and construction of an `rMix` object) and `plot.Mix` for the plot method.

`Mix` for the construction of `Mix` objects, `dMix` for the density of a mixture.

Examples

```
# define 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

# plot 'Mix' object
plot(normLocMix)
plot(poisMix)
```


```
# define 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

# plot 'Mix' object
plot(normLocMix)
plot(poisMix)
```


---

mix.lrt *Estimate a Mixture's Complexity Based on Likelihood Ratio Test Statistics*

Description

Estimation of a mixture's the complexity as well as its component weights and parameters based on comparing the likelihood ratio test statistic (LRTS) to a bootstrapped quantile.

Usage

```
mix.lrt(obj, j.max = 10, B = 100, quantile = 0.95, control = c(trace = 0), ...)
```

Arguments

obj	object of class datMix.
j.max	integer giving the maximal complexity to be considered.
B	integer specifying the number of bootstrap replicates.
quantile	numeric between 0 and 1 specifying the bootstrap quantile to which the observed LRTS will be compared.
control	control list of optimization parameters, see solnp.
...	further arguments passed to the boot function.

Details

Define the *complexity* of a finite mixture F as the smallest integer p, such that its pdf/pmf f can be written as

$$f(x) = w_1 * g(x; \theta_1) + \ldots + w_p * g(x; \theta_p).$$

To estimate p, mix.lrt sequentially tests $p = j$ versus $p = j + 1$ for $j = 1, 2, \ldots$, by finding the maximum likelihood estimator (MLE) for the density of a mixture with j and $j + 1$ components and calculating the corresponding likelihood ratio test statistic (LRTS). Next, a parametric bootstrap procedure is used to generate B samples of size n from a j-component mixture given the previously calculated MLE. For each of the bootstrap samples, the MLEs corresponding to densities of mixtures with j and $j + 1$ components are calculated, as well as the LRTS. The null hypothesis $H_0 : p = j$ is rejected and j increased by 1 if the LRTS based on the original data is larger than the chosen quantile of its bootstrapped counterparts. Otherwise, j is returned as the complexity estimate. The MLEs are calculated via the MLE.function attribute (of the datMix object obj) for $j = 1$, if it is supplied. For all other j (and also for $j = 1$ in case MLE.function = NULL) the solver solnp is used to calculate the minimum of the negative log likelihood. The initial values supplied to the solver are calculated as follows: the data is clustered into j groups by the function clara and the data corresponding to each group is given to MLE.function (if supplied to the datMix object, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights and the MLE's are taken as initial component parameter estimates.

Value

Object of class paramEst with the following attributes:

dat	data based on which the complexity is estimated.
dist	character string stating the (abbreviated) name of the component distribution, such that the function ddist evaluates its density function and rdist generates random numbers.
ndistparams	integer specifying the number of parameters identifying the component distribution, i.e. if θ is in R^d then ndistparams$= d$.

formals.dist	string vector specifying the names of the formal arguments identifying the distribution dist and used in ddist and rdist, e.g. for a gaussian mixture (dist = norm) amounts to mean and sd, as these are the formal arguments used by dnorm and rnorm.
discrete	logical indicating whether the underlying mixture distribution is discrete.
mle.fct	attribute MLE.function of obj.
pars	Say the complexity estimate is equal to some j. Then pars is a numeric vector of size $(d+1)*j-1$ specifying the component weight and parameter estimates, given as $$(w_1, ...w_{j-1}, \theta 1_1, ...\theta 1_j, \theta 2_1, ...\theta d_j).$$
values	numeric vector of function values gone through during optimization at iteration j, the last entry being the value at the optimum.
convergence	indicates whether the solver has converged (0) or not (1 or 2) at iteration j.

See Also

solnp for the solver, datMix for the creation of the datMix object.

Examples

```
### generating 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

### generating 'rMix' from 'Mix' object (with 1000 observations)
set.seed(0)
normLocRMix <- rMix(1000, normLocMix)


### generating 'datMix' from 'R' object

## generate list of parameter bounds

norm.bound.list <- vector(mode = "list", length = 2)
names(norm.bound.list) <- c("mean", "sd")
norm.bound.list$mean <- c(-Inf, Inf)
norm.bound.list$sd <- c(0, Inf)

## generate MLE functions

# for "mean"
MLE.norm.mean <- function(dat) mean(dat)
# for "sd" (the sd function uses (n-1) as denominator)
MLE.norm.sd <- function(dat){
sqrt((length(dat) - 1) / length(dat)) * sd(dat)
}
# combining the functions to a list
MLE.norm.list <- list("MLE.norm.mean" = MLE.norm.mean,
                      "MLE.norm.sd" = MLE.norm.sd)
```

```
## generating 'datMix' object
normLoc.dM <- RtoDat(normLocRMix, theta.bound.list = norm.bound.list,
                     MLE.function = MLE.norm.list)

### complexity and parameter estimation

set.seed(0)
res <- mix.lrt(normLoc.dM, B = 30)
plot(res)
```

---

nonparamHankel *Estimate Mixture Complexity Based on Hankel Matrix*

Description

Estimation of mixture complexity based on estimating the determinant of the Hankel matrix of the moments of the mixing distribution. The estimated determinants can be scaled and/or penalized.

Usage

```
nonparamHankel(obj, j.max = 10, pen.function = NULL, scaled = FALSE, B = 1000, ...)

## S3 method for class 'hankDet'
print(x, ...)

## S3 method for class 'hankDet'
plot(
  x,
  type = "b",
  xlab = "j",
  ylab = NULL,
  mar = NULL,
  ylim = c(min(0, min(obj)), max(obj)),
  ...
)
```

Arguments

obj	object of class datMix.
j.max	integer specifying the maximal number of components to be considered.
pen.function	a function with arguments j and n specifying the penalty added to the determinant value given sample size n and the currently assumed complexity j. If left empty no penalty will be added. If non-empty and scaled is TRUE, the penalty function will be added after the determinants are scaled.
scaled	logical specifying whether the vector of estimated determinants should be scaled.

B	integer specifying the number of bootstrap replicates used for scaling of the determinants. Ignored if scaled is FALSE.
...	**in** nonparamHankel(): further arguments passed to the boot function if scaled is TRUE. **in** plot.hankDet(): further arguments passed to plot. **in** print.hankDet(): further arguments passed to print.
x	object of class hankDet.
type	character denoting type of plot, see, e.g. lines. Defaults to "b".
xlab, ylab	labels for the x and y axis with defaults (the default for ylab is created within the function, if no value is supplied).
mar	numerical vector of the form c(bottom, left, top, right) which gives the number of lines of margin to be specified on the four sides of the plot, see par.
ylim	range of y values to use.

Details

Define the *complexity* of a finite mixture F as the smallest integer p, such that its pdf/pmf f can be written as

$$f(x) = w_1 * g(x; \theta_1) + \ldots + w_p * g(x; \theta_p).$$

nonparamHankel estimates p by iteratively increasing the assumed complexity j and calculating the determinant of the $(j+1)$x$(j+1)$ Hankel matrix made up of the first $2j$ raw moments of the mixing distribution. As shown by Dacunha-Castelle & Gassiat (1997), once the correct complexity is reached (i.e. for all $j >= p$), this determinant is zero. This suggests an estimation procedure for p based on initially finding a consistent estimator of the moments of the mixing distribution and then choosing the estimator $estim_p$ as the value of j which yields a sufficiently small value of the determinant. Since the estimated determinant is close to 0 for all $j >= p$, this could lead to choosing $estim_p$ rather larger than the true value. The function therefore returns all estimated determinant values corresponding to complexities up to j.max, so that the user can pick the lowest j generating a sufficiently small determinant. In addition, the function allows the inclusion of a penalty term as a function of the sample size n and the currently assumed complexity j which will be added to the determinant value (by supplying pen.function), and/or scaling of the determinants (by setting scaled = TRUE). For scaling, a nonparametric bootstrap is used to calculate the covariance of the estimated determinants, with B being the size of the bootstrap sample. The inverse of the square root of this covariance matrix (i.e. the matrix $S^{(-1)}$ such that $A = SS^T$, where A is the covariance matrix) is then multiplied with the estimated determinant vector to get the scaled determinant vector. For a thorough discussion of the methods that can be used for the estimation of the moments see the details section of datMix.

Value

The vector of estimated determinants (optionally scaled and/or penalized), given back as an object of class hankDet with the following attributes:

scaled	logical indicating whether the determinants are scaled.
pen	logical indicating whether a penalty was added to the determinants.
dist	character string stating the (abbreviated) name of the component distribution, such that the function ddist evaluates its density function and rdist generates random numbers.

References

D. Dacunha-Castelle and E. Gassiat, "The estimation of the order of a mixture model", Bernoulli, Volume 3, Number 3, 279-299, 1997.

See Also

paramHankel for a similar approach which estimates the component weights and parameters on top of the complexity, datMix for the creation of the datMix object.

Examples

```
## create 'Mix' object
geomMix <- Mix("geom", w = c(0.1, 0.6, 0.3), prob = c(0.8, 0.2, 0.4))

## create random data based on 'Mix' object (gives back 'rMix' object)
set.seed(1)
geomRMix <- rMix(1000, obj = geomMix)

## create 'datMix' object for estimation

# explicit function giving the estimate for the j^th moment of the
# mixing distribution, needed for Hankel.method "explicit"

explicit.fct.geom <- function(dat, j){
 1 - ecdf(dat)(j - 1)
}

## generating 'datMix' object
geom.dM <- RtoDat(geomRMix, Hankel.method = "explicit",
                  Hankel.function = explicit.fct.geom)

## function for penalization
pen <- function(j, n){
  (j*log(n))/(sqrt(n))
}

## estimate determinants
set.seed(1)
geomdets_pen <- nonparamHankel(geom.dM, pen.function = pen, j.max = 5)
plot(geomdets_pen, main = "Three component geometric mixture")
```

---

paramHankel *Estimate a Mixture's Complexity (and Component Weights/Parameters) Based on Hankel Matrix*

Description

Estimation method of mixture complexity as well as component weights and parameters based on estimating the determinant of the Hankel matrix of the moments of the mixing distribution and comparing it to determinant values generated by a parametric bootstrap.

Usage

```
paramHankel(obj, j.max = 10, B = 1000, ql = 0.025, qu = 0.975,
                                control = c(trace = 0), ...)

paramHankel.scaled(obj, j.max = 10, B = 100, ql = 0.025, qu = 0.975,
                                     control = c(trace = 0), ...)

## S3 method for class 'paramEst'
plot(x, mixture = TRUE, components = TRUE, ylim = NULL, cex.main = 0.9, ...)

## S3 method for class 'paramEst'
print(x, ...)
```

Arguments

obj	object of class datMix.
j.max	integer stating the maximal number of components to be considered.
B	integer specifying the number of bootstrap replicates.
ql	numeric between 0 and 1 specifying the lower bootstrap quantile to which the observed determinant value will be compared.
qu	numeric between 0 and 1 specifying the upper bootstrap quantile to which the observed determinant value will be compared.
control	control list of optimization parameters, see solnp.
...	**in** paramHankel() **and** paramHankel.scaled(): further arguments passed to the boot function. **in** plot.hankDet(): further arguments passed to the hist function plotting the data. **in** print.hankDet(): further arguments passed to the printCoefmat function.
x	object of class paramEst.
mixture	logical indicating whether the estimated mixture density should be drawn, set to TRUE by default.
components	logical indicating whether the individual mixture components should be drawn, set to TRUE by default.
ylim	range of y values to use; if not specified (or containing NA), the function tries to construct reasonable default values itself.
cex.main	The magnification to be used for main titles relative to the current setting of cex, see par.

Details

Define *complexity* of a finite mixture F as the smallest integer p, such that its pdf/pmf f can be written as

$$f(x) = w_1 * g(x; \theta_1) + \ldots + w_p * g(x; \theta_p).$$

The `paramHankel` procedure initially assumes the mixture to only contain a single component, setting $j = 1$, and then sequentially tests $p = j$ versus $p = j + 1$ for $j = 1, 2, \ldots$, until the algorithm terminates. To do so, it determines the MLE for a j-component mixture, generates `B` parametric bootstrap samples of size n from the distribution the MLE corresponds to and calculates `B` determinants of the corresponding $(j+1)x(j+1)$ Hankel matrices of the first $2j$ raw moments of the mixing distribution (for details see `nonparamHankel`). The null hypothesis $H_0 : p = j$ is rejected and j increased by 1 if the determinant value based on the original data lies outside of the interval $[ql, qu]$, a range specified by the `ql` and `qu` empirical quantiles of the bootstrapped determinants. Otherwise, j is returned as the complexity estimate. `paramHankel.scaled` functions similarly to `paramHankel` with the exception that the bootstrapped determinants are scaled by the empirical standard deviation of the bootstrap sample. To scale the original determinant, `B` nonparametric bootstrap samples of size n are generated from the data, the corresponding determinants are calculated and their empirical standard deviation is used. The MLEs are calculated via the `MLE.function` attribute (of the `datMix` object `obj`) for $j = 1$, if it is supplied. For all other j (and also for $j = 1$ in case `MLE.function = NULL`) the solver `solnp` is used to calculate the minimum of the negative log likelihood. The initial values supplied to the solver are calculated as follows: the data is clustered into j groups by the function `clara` and the data corresponding to each group is given to `MLE.function` (if supplied to the `datMix` object, otherwise numerical optimization is used here as well). The size of the groups is taken as initial component weights and the MLE's are taken as initial component parameter estimates.

Value

Object of class `paramEst` with the following attributes:

`dat`	data based on which the complexity is estimated.
`dist`	character string stating the (abbreviated) name of the component distribution, such that the function `ddist` evaluates its density function and `rdist` generates random numbers.
`ndistparams`	integer specifying the number of parameters identifying the component distribution, i.e. if θ is in R^d then `ndistparams`$= d$.
`formals.dist`	string vector specifying the names of the formal arguments identifying the distribution `dist` and used in `ddist` and `rdist`, e.g. for a gaussian mixture (`dist = norm`) amounts to `mean` and `sd`, as these are the formal arguments used by `dnorm` and `rnorm`.
`discrete`	logical indicating whether the underlying mixture distribution is discrete.
`mle.fct`	attribute `MLE.function` of `obj`.
`pars`	Say the complexity estimate is equal to some j. Then `pars` is a numeric vector of size $(d+1)*j-1$ specifying the component weight and parameter estimates, given as $(w_1, \ldots w_{j-1}, \theta 1_1, \ldots \theta 1_j, \theta 2_1, \ldots \theta d_j)$.

values	numeric vector of function values gone through during optimization at iteration j, the last entry being the value at the optimum.
convergence	indicates whether the solver has converged (0) or not (1 or 2) at iteration j.

See Also

nonparamHankel for estimation of the mixture complexity based on the Hankel matrix without parameter estimation, solnp for the solver, datMix for creation of the datMix object.

Examples

```
## create 'Mix' object
poisMix <- Mix("pois", w = c(0.45, 0.45, 0.1), lambda = c(1, 5, 10))

## create random data based on 'Mix' object (gives back 'rMix' object)
set.seed(1)
poisRMix <- rMix(1000, obj = poisMix)

## create 'datMix' object for estimation
# generate list of parameter bounds
poisList <- vector(mode = "list", length = 1)
names(poisList) <- "lambda"
poisList$lambda <- c(0, Inf)

# generate MLE function
MLE.pois <- function(dat){
  mean(dat)
}

# generate function needed for estimating the j^th moment of the
# mixing distribution via Hankel.method "explicit"

explicit.pois <- function(dat, j){
  res <- 1
  for (i in 0:(j-1)){
    res <- res*(dat-i)
  }
  return(mean(res))
}

# generating 'datMix' object
pois.dM <- RtoDat(poisRMix, theta.bound.list = poisList, MLE.function = MLE.pois,
                  Hankel.method = "explicit", Hankel.function = explicit.pois)


## complexity and parameter estimation

set.seed(1)
res <- paramHankel(pois.dM)
plot(res)
```

rMix *Generate a Random Sample from a Mixture Distribution*

Description

Generate a random sample of size n, distributed according to a mixture specified as Mix object. Returns an object of class rMix.

plot method for rMix objects, plotting the histogram of the random sample, with the option of additionally plotting the components (stacked or plotted over one another).

Usage

```
rMix(n, obj)

is.rMix(x)

## S3 method for class 'rMix'
plot(
  x,
  xlab = attr(obj, "name"),
  ylim = NULL,
  main = paste("Histogram of", attr(obj, "name")),
  breaks = NULL,
  col = "grey",
  components = TRUE,
  stacked = FALSE,
  component.colors = NULL,
  freq = TRUE,
  plot = TRUE,
  ...
)

## S3 method for class 'rMix'
print(x, ...)
```

Arguments

n	integer specifying the number of observations.
obj	object of class Mix.
x	object of class rMix.
xlab	label for the x axis with default.
ylim	range of y values to use; if not specified (or containing NA), default values are used.

main	main title of the plot with default.
breaks	see hist. If left unspecified the function tries to construct reasonable default values.
col	a colour to be used to fill the bars of the histogram evaluated on the whole data.
components	logical indicating whether the plot should show to which component the observations belong (either by plotting individual histograms or by overlaying a stacked barplot), defaulting to TRUE. Ignored if plot is FALSE.
stacked	logical indicating whether the component plots should be stacked or plotted over one another, defaulting to FALSE. Ignored if components is FALSE or ignored itself.
component.colors	the colors for the component plots. If left unspecified default colors are used.
freq	logical, if TRUE, the histogram graphic is a representation of frequencies, if FALSE, probability densities. See hist.
plot	logical, if TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned. See hist.
...	further arguments passed to the histogram function evaluated on the whole data as well as the component data (if components is TRUE and stacked is FALSE).

Details

For a mixture of p components, generate the number of observations in each component as multinomial, and then use an implemented random variate generation function for each component. The integer (multinomial) numbers are generated via sample.

Value

An object of class rMix with the following attributes (for further explanations see Mix):

name	name of the Mix object that was given as input.
dist	character string stating the (abbreviated) name of the component distribution, such that the function ddist evaluates its density function and rdist generates random numbers.
discrete	logical indicating whether the underlying mixture distribution is discrete.
theta.list	named list specifying the parameter values of the p components.
w	numeric vector of length p, specifying the mixture weights $w[i]$ of the components, $i = 1, \ldots, p$.
indices	numeric vector of length n containing integers between 1 and p specifying which mixture component each observation belongs to.

See Also

dMix for the density, Mix for the construction of Mix objects and plot.rMix for the plot method.

rMix for the creation of rMix objects.

Examples

```
# define 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

# generate n random samples
set.seed(1)
x <- rMix(1000, normLocMix)
hist(x)


# define 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

# generate n random samples
set.seed(1)
x <- rMix(1000, normLocMix)
plot(x)
```

RtoDat *Converting* rMix *to* datMix *Objects*

Description

Converting an object of class rMix to an object of class datMix, so that it can be passed to functions estimating the mixture complexity.

Usage

```
RtoDat(obj, theta.bound.list = NULL, MLE.function = NULL, Hankel.method = NULL,
                                                          Hankel.function = NULL)
```

Arguments

obj	object of class rMix.
theta.bound.list	a named list specifying the upper and the lower bound for the component parameters. The names of the list elements have to match the names of the formal arguments of the functions ddist and rdist exactly (the dist attribute is specified when creating the Mix object obj). For a gaussian mixture with dist = norm, the list elements would have to be named mean and sd, as these are the formal arguments used by rnorm and dnorm. Has to be supplied if a method that estimates the component weights and parameters is to be used.
MLE.function	function (or list of functions) which takes as input the data and gives as output the maximum likelihood estimator for the parameter(s) of a one component mixture (i.e. the standard MLE of the component distribution dist). If the component distribution has more than one parameter, a list of functions has to

be supplied and the order of the MLE functions has to match the order of the component parameters in theta.bound.list (e.g. for a normal mixture, if the first entry of theta.bound.list is the bounds of the mean, then then first entry of MLE.function has to be the MLE of the mean). If this argument is supplied and the datMix object is handed over to a complexity estimation procedure relying on optimizing over a likelihood function, the MLE.function attribute will be used for the single component case. In case the objective function is either not a likelihood or corresponds to a mixture with more than 1 components, numerical optimization will be used based on Rsolnp's function solnp, but MLE.function will be used to calculate the initial values passed to solnp. Specifying MLE.function is optional and if it is not, for example because the MLE solution does not exists in closed form, numerical optimization is used to find the relevant MLE's.

Hankel.method character string in c("explicit","translation","scale"), specifying the method of estimating the moments of the mixing distribution used to calculate the relevant Hankel matrix. Has to be specified when using nonparamHankel, paramHankel or paramHankel.scaled. For further details see the details section of datMix.

Hankel.function function needed for the moment estimation via Hankel.method. This normally depends on Hankel.method as well as dist. For further details see the details section of datMix.

Value

An object of class datMix with the following attributes (for further explanations see datMix):

dist

discrete
theta.bound.list

MLE.function

Hankel.method
Hankel.function

See Also

datMix for direct generation of a datMix object from a vector of observations.

Examples

```
### generating 'Mix' object
normLocMix <- Mix("norm", w = c(0.3, 0.4, 0.3), mean = c(10, 13, 17), sd = c(1, 1, 1))

### generating 'rMix' from 'Mix' object (with 1000 observations)
set.seed(1)
normLocRMix <- rMix(1000, normLocMix)
```

```
### generating 'datMix' from 'R' object

## generate list of parameter bounds

norm.bound.list <- vector(mode = "list", length = 2)
names(norm.bound.list) <- c("mean", "sd")
norm.bound.list$mean <- c(-Inf, Inf)
norm.bound.list$sd <- c(0, Inf)

## generate MLE functions

# for "mean"
MLE.norm.mean <- function(dat) mean(dat)
# for "sd" (the sd function uses (n-1) as denominator)
MLE.norm.sd <- function(dat){
  sqrt((length(dat) - 1) / length(dat)) * sd(dat)
}
# combining the functions to a list
MLE.norm.list <- list("MLE.norm.mean" = MLE.norm.mean,
                      "MLE.norm.sd" = MLE.norm.sd)

## function giving the j^th raw moment of the standard normal distribution,
## needed for calculation of the Hankel matrix via the "translation" method
## (assuming gaussian components with variance 1)

mom.std.norm <- function(j){
 ifelse(j %% 2 == 0, prod(seq(1, j - 1, by = 2)), 0)
}

normLoc.dM <- RtoDat(normLocRMix, theta.bound.list = norm.bound.list,
                     MLE.function = MLE.norm.list, Hankel.method = "translation",
                     Hankel.function = mom.std.norm)

### using 'datMix' object to estimate the mixture

set.seed(0)
res <- paramHankel.scaled(normLoc.dM)
plot(res)
```

Index