Package 'abn'

May 7, 2026

Title Modelling Multivariate Data with Additive Bayesian Networks

Version 3.1.13

Date 2025-12-18

Description The 'abn' R package facilitates Bayesian network analysis, a probabilistic graphical model that derives from empirical data a directed acyclic graph (DAG). This DAG describes the dependency structure between random variables. The R package 'abn' provides routines to help determine optimal Bayesian network models for a given data set. These models are used to identify statistical dependencies in messy, complex data. Their additive formulation is equivalent to multivariate generalised linear modelling, including mixed models with independent and identically distributed (iid) random effects. The core functionality of the 'abn' package revolves around model selection, also known as structure discovery. It supports both exact and heuristic structure learning algorithms and does not restrict the data distribution of parent-child combinations, providing flexibility in model creation and analysis. The 'abn' package uses Laplace approximations for metric estimation and includes wrappers to the 'INLA' package. It also employs 'JAGS' for data simulation purposes. For more resources and information, visit the 'abn' website.

License GPL (>= 3)

URL https://r-bayesian-networks.org/, https://github.com/furrer-lab/abn

BugReports https://github.com/furrer-lab/abn/issues

Depends R (>= 4.0.0)

Imports doParallel, foreach, glmmTMB, graph, jsonlite, lme4, mclogit, methods, nnet, Rcpp, Rgraphviz, rjags, stringi

Suggests bookdown, boot, brglm, devtools (>= 2.4.5), ggplot2, gridExtra, INLA, knitr, Matrix, MatrixModels (>= 0.5.3), microbenchmark, R.rsp, RhpcBLASctl, rmarkdown, testthat (>= 3.0.0), entropy, moments, R6

LinkingTo Rcpp, RcppArmadillo

VignetteBuilder knitr

Additional_repositories https://inla.r-inla-download.org/R/stable/

Config/testthat/edition 3

Encoding UTF-8

LazyData TRUE

RoxygenNote 7.3.3

SystemRequirements pkg-config, cmake, gsl, jpeg, gdal, geos, proj, udunits-2, openssl, libcurl, jags

NeedsCompilation yes

Author Matteo Delucchi [aut, cre] (ORCID: <https://orcid.org/0000-0002-9327-1496>), Reinhard Furrer [aut] (ORCID: <https://orcid.org/0000-0002-6319-2332>), Gilles Kratzer [aut] (ORCID: <https://orcid.org/0000-0002-5929-8935>), Fraser Iain Lewis [aut] (ORCID: <https://orcid.org/0000-0003-4580-2712>), Jonas I. Liechti [ctb] (ORCID: <https://orcid.org/0000-0003-3447-3060>), Marta Pittavino [ctb] (ORCID: <https://orcid.org/0000-0002-1232-1034>), Kalina Cherneva [ctb]

Maintainer Matteo Delucchi <matteo.delucchi@math.uzh.ch>

Repository CRAN

Date/Publication 2025-12-19 10:20:26 UTC

Contents

`AIC.abnFit` *Print AIC of objects of class* `abnFit`

Description

Print AIC of objects of class `abnFit`

Usage

```
## S3 method for class 'abnFit'
AIC(object, digits = 3L, verbose = TRUE, ...)
```

Arguments

object	Object of class abnFit
digits	number of digits of the results.
verbose	print additional output.
...	additional parameters. Not used at the moment.

Value

prints the AIC of the fitted model.

---

as.data.frame.abnDag *Transform the adjacency matrix representation of a DAG to a data.frame with columns "from" and "to" representing directed edges.*

Description

Transform the adjacency matrix representation of a DAG to a data.frame with columns "from" and "to" representing directed edges.

Usage

```
## S3 method for class 'abnDag'
as.data.frame(x, ...)
```

Arguments

x	An object of class abnDag
...	Additional arguments (currently unused)

Details

The adjacency matrix in abnDag objects has parents in columns and children in rows. A value of 1 at position (i,j) indicates an arc from parent j to child i.

Value

A data.frame with columns "from" and "to" representing directed edges from parent nodes (from) to child nodes (to)

Examples

```
# Create example DAG matrix
mydag <- createAbnDag(dag = ~a+b|a, data.df = data.frame("a"=1, "b"=1))

# Convert to edge list
as.data.frame(mydag)
```

BIC.abnFit *Print BIC of objects of class* abnFit

Description

Print BIC of objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
BIC(object, digits = 3L, verbose = TRUE, ...)
```

Arguments

object	Object of class abnFit
digits	number of digits of the results.
verbose	print additional output.
...	additional parameters. Not used at the moment.

Value

prints the BIC of the fitted model.

build.control *Control the iterations in* buildScoreCache

Description

Allow the user to set restrictions in the buildScoreCache for both the Bayesian and the MLE approach. Control function similar to fit.control.

Usage

```
build.control(
  method = "bayes",
  max.mode.error = 10,
  mean = 0,
  prec = 0.001,
  loggam.shape = 1,
  loggam.inv.scale = 5e-05,
  max.iters = 100,
  epsabs = 1e-07,
  error.verbose = FALSE,
  trace = 0L,
  epsabs.inner = 1e-06,
```

```
  max.iters.inner = 100,
  finite.step.size = 1e-07,
  hessian.params = c(1e-04, 0.01),
  max.iters.hessian = 10,
  max.hessian.error = 0.5,
  factor.brent = 100,
  maxiters.hessian.brent = 100,
  num.intervals.brent = 100,
  n.grid = 250,
  ncores = 1,
  cluster.type = "FORK",
  max.irls = 100,
  tol = 1e-08,
  tolPwrss = 1e-07,
  check.rankX = "message+drop.cols",
  check.scaleX = "message+rescale",
  check.conv.grad = "message",
  check.conv.singular = "message",
  check.conv.hess = "message",
  xtol_abs = 1e-06,
  ftol_abs = 1e-06,
  trace.mblogit = FALSE,
  catcov.mblogit = "free",
  epsilon = 1e-06,
  only_glmmTMB_poisson = FALSE,
  seed = 9062019L
)
```

Arguments

method	a character that takes one of two values: "bayes" or "mle". Overrides method argument from buildScoreCache.
max.mode.error	if the estimated modes from INLA differ by a factor of max.mode.error or more from those computed internally, then results from INLA are replaced by those computed internally. To force INLA always to be used, then max.mode.error=100, to force INLA never to be used max.mod.error=0. See also fitAbn.
mean	the prior mean for all the Gaussian additive terms for each node. INLA argument control.fixed=list(mean.intercept=...) and control.fixed=list(mean=...).
prec	the prior precision ($\tau = \frac{1}{\sigma^2}$) for all the Gaussian additive term for each node. INLA argument control.fixed=list(prec.intercept=...) and control.fixed=list(prec=...).
loggam.shape	the shape parameter in the Gamma distribution prior for the precision in a Gaussian node. INLA argument control.family=list(hyper = list(prec = list(prior="loggamma",pa loggam.inv.scale))).
loggam.inv.scale	the inverse scale parameter in the Gamma distribution prior for the precision in a Gaussian node. INLA argument control.family=list(hyper = list(prec = list(prior="loggamma",param=c(loggam.shape, loggam.inv.scale)))).

`max.iters` total number of iterations allowed when estimating the modes in Laplace approximation. passed to `.Call("fit_single_node", ...)`.

`epsabs` absolute error when estimating the modes in Laplace approximation for models with no random effects. Passed to `.Call("fit_single_node", ...)`.

`error.verbose` logical, additional output in the case of errors occurring in the optimization. Passed to `.Call("fit_single_node", ...)`.

`trace` Non-negative integer. If positive, tracing information on the progress of the "L-BFGS-B" optimization is produced. Higher values may produce more tracing information. (There are six levels of tracing. To understand exactly what these do see the source code.). Passed to `.Call("fit_single_node", ...)`.

`epsabs.inner` absolute error in the maximization step in the (nested) Laplace approximation for each random effect term. Passed to `.Call("fit_single_node", ...)`.

`max.iters.inner` total number of iterations in the maximization step in the nested Laplace approximation. Passed to `.Call("fit_single_node", ...)`.

`finite.step.size` suggested step length used in finite difference estimation of the derivatives for the (outer) Laplace approximation when estimating modes. Passed to `.Call("fit_single_node", ...)`.

`hessian.params` a numeric vector giving parameters for the adaptive algorithm, which determines the optimal stepsize in the finite-difference estimation of the hessian. First entry is the initial guess, second entry absolute error. Passed to `.Call("fit_single_node", ...)`.

`max.iters.hessian` integer, maximum number of iterations to use when determining an optimal finite difference approximation (Nelder-Mead). Passed to `.Call("fit_single_node", ...)`.

`max.hessian.error` if the estimated log marginal likelihood when using an adaptive 5pt finite-difference rule for the Hessian differs by more than `max.hessian.error` from when using an adaptive 3pt rule then continue to minimize the local error by switching to the Brent-Dekker root bracketing method. Passed to `.Call("fit_single_node", ...)`.

`factor.brent` if using Brent-Dekker root bracketing method then define the outer most interval end points as the best estimate of h (stepsize) from the Nelder-Mead as $h/factor.brent, h * factor.brent)$. Passed to `.Call("fit_single_node", ...)`.

`maxiters.hessian.brent` maximum number of iterations allowed in the Brent-Dekker method. Passed to `.Call("fit_single_node", ...)`.

`num.intervals.brent` the number of initial different bracket segments to try in the Brent-Dekker method. Passed to `.Call("fit_single_node", ...)`.

`n.grid` recompute density on an equally spaced grid with `n.grid` points.

`ncores` The number of cores to parallelize to, see 'Details'. If >0, the number of CPU cores to be used. -1 for all available -1 core. Only for `method="mle"`.

`cluster.type` The type of cluster to be used, see `?parallel::makeCluster`. `abn` then defaults to `"PSOCK"` on Windows and `"FORK"` on Unix-like systems. With "FORK" the child process are started with `rscript_args = "--no-environ"` to avoid loading the whole workspace into each child.

`max.irls` total number of iterations for estimating network scores using an Iterative Reweighed Least Square algorithm. Is this DEPRECATED?

`tol` real number giving the minimal tolerance expected to terminate the Iterative Reweighed Least Square algorithm to estimate network score. Passed to `irls_binomial_cpp_fast_br` and `irls_poisson_cpp_fast`.

`tolPwrss` numeric scalar passed to `glmerControl` - the tolerance for declaring convergence in the penalized iteratively weighted residual sum-of-squares step. Similar to `tol`.

`check.rankX` character passed to `lmerControl` and `glmerControl` - specifying if `rankMatrix(X)` should be compared with `ncol(X)` and if columns from the design matrix should possibly be dropped to ensure that it has full rank. Defaults to `message+drop.cols`.

`check.scaleX` character passed to `lmerControl` and `glmerControl` - check for problematic scaling of columns of fixed-effect model matrix, e.g. parameters measured on very different scales. Defaults to `message+rescale`.

`check.conv.grad`
character passed to `lmerControl` and `glmerControl` - checking the gradient of the deviance function for convergence. Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

`check.conv.singular`
character passed to `lmerControl` and `glmerControl` - checking for a singular fit, i.e. one where some parameters are on the boundary of the feasible space (for example, random effects variances equal to 0 or correlations between random effects equal to +/- 1.0). Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

`check.conv.hess`
character passed to `lmerControl` and `glmerControl` - checking the Hessian of the deviance function for convergence. Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

`xtol_abs` Defaults to 1e-6 stop on small change of parameter value. Only for `method='mle'`, `group.var=...`. Default convergence tolerance for fitted `(g)lmer` models is reduced to the value provided here if default values did not fit. This value here is passed to the `optCtrl` argument of `(g)lmer` (see help of `lme4::convergence()`).

`ftol_abs` Defaults to 1e-6 stop on small change in deviance. Similar to `xtol_abs`.

`trace.mblogit` logical indicating if output should be produced for each iteration. Directly passed to `trace` argument in `mclogit.control`. Is independent of `verbose`.

`catcov.mblogit` Defaults to "free" meaning that there are no restrictions on the covariances of random effects between the logit equations. Set to "diagonal" if random effects pertinent to different categories are uncorrelated or "single" if random effect variances pertinent to all categories are identical.

epsilon — Defaults to 1e-8. Positive convergence tolerance ϵ that is directly passed to the `control` argument of `mclogit::mblogit` as `mclogit.control`. Only for `method='mle'`, `group.var=...`.

only_glmmTMB_poisson — logical, if TRUE only use `glmmTMB` to fit Poisson nodes with random effects. This is useful if `glmer` fails due to convergence issues. Default is FALSE.

seed — a non-negative integer which sets the seed in `set.seed(seed)`.

Details

Parallelization over all children is possible via the function `foreach` of the package **doParallel**. `ncores=0` or `ncores=1` use single threaded `foreach`. `ncores=-1` uses all available cores but one.

Value

Named list according the provided arguments.

See Also

`fit.control`.

Other buildScoreCache: `buildScoreCache()`

Examples

```
ctrlmle <- abn::build.control(method = "mle",
                         ncores = 0,
                         cluster.type = "PSOCK",
                         max.irls = 100,
                         tol = 10^-11,
                         tolPwrss = 1e-7,
                         check.rankX = "message+drop.cols",
                         check.scaleX = "message+rescale",
                         check.conv.grad = "message",
                         check.conv.singular = "message",
                         check.conv.hess = "message",
                         xtol_abs = 1e-6,
                         ftol_abs = 1e-6,
                         trace.mblogit = FALSE,
                         catcov.mblogit = "free",
                         epsilon = 1e-6,
                         only_glmmTMB_poisson=FALSE,
                         seed = 9062019L)
ctrlbayes <- abn::build.control(method = "bayes",
                            max.mode.error = 10,
                            mean = 0, prec = 0.001,
                            loggam.shape = 1,
                            loggam.inv.scale = 5e-05,
                            max.iters = 100,
                            epsabs = 1e-07,
                            error.verbose = FALSE,
                            epsabs.inner = 1e-06,
```

```
                    max.iters.inner = 100,
                    finite.step.size = 1e-07,
                    hessian.params = c(1e-04, 0.01),
                    max.iters.hessian = 10,
                    max.hessian.error = 0.5,
                    factor.brent = 100,
                    maxiters.hessian.brent = 100,
                    num.intervals.brent = 100,
                    tol = 10^-8,
                    seed = 9062019L)
```

`check.valid.fitControls` *Simple check on the control parameters*

Description

Simple check on the control parameters

Usage

```
check.valid.fitControls(control, method = "bayes", verbose = FALSE)
```

Arguments

`control`	list of control arguments with new parameters supplied to `buildScoreCache` or `fitAbn`.
`method`	"bayes" or "mle" strategy from argument `method=...` in `buildScoreCache` or `fitAbn`. Defaults to "bayes".
`verbose`	when TRUE additional information is printed. Defaults to FALSE.

Value

list with all control arguments with respect to the method but with new values.

`coef.abnFit` *Print coefficients of objects of class* `abnFit`

Description

Print coefficients of objects of class `abnFit`

Usage

```
## S3 method for class 'abnFit'
coef(object, digits = 3L, verbose = TRUE, ...)
```

Arguments

object	Object of class abnFit
digits	number of digits of the results.
verbose	print additional output.
...	additional parameters. Not used at the moment.

Value

prints the coefficients of the fitted model.

---

compareDag *Compare two DAGs or EGs*

Description

Function that returns multiple graph metrics to compare two DAGs or essential graphs, known as confusion matrix or error matrix.

Usage

```
compareDag(ref, test, node.names = NULL, checkDAG = TRUE)
```

Arguments

ref	a matrix or a formula statement (see details for format) defining the reference network structure, a directed acyclic graph (DAG). Note that row names must be set or given in node.names if the DAG is given via a formula statement.
test	a matrix or a formula statement (see details for format) defining the test network structure, a directed acyclic graph (DAG). Note that row names must be set or given in node.names if the DAG is given via a formula statement.
node.names	a vector of names if the DAGs are given via formula, see details.
checkDAG	should the DAGs be tested for DAGs (default).

Details

This R function returns standard Directed Acyclic Graph comparison metrics. In statistical classification, those metrics are known as a confusion matrix or error matrix.

Those metrics allows visualization of the difference between different DAGs. In the case where comparing TRUTH to learned structure or two learned structures, those metrics allow the user to estimate the performance of the learning algorithm. In order to compute the metrics, a contingency table is computed of a pondered difference of the adjacency matrices od the two graphs.

The ref or test can be provided using a formula statement (similar to GLM input). A typical formula is ~ node1|parent1:parent2 + node2:node3|parent3. The formula statement have to start with ~. In this example, node1 has two parents (parent1 and parent2). node2 and node3 have the same parent3. The parents names have to exactly match those given in node.names. : is the

separtor between either children or parents, | separates children (left side) and parents (right side), + separates terms, . replaces all the variables in node.names.

To test for essential graphs (or graphs) in general, the test for DAG need to be switched off checkDAG=FALSE. The function compareEG() is a wrapper to compareDag(, checkDAG=FALSE).

Value

TP True Positive

TN True Negative

FP False Positive

FN False Negative

CP Condition Positive (ref)

CN Condition Negative (ref)

PCP Predicted Condition Positive (test)

PCN Predicted Condition Negative (test)

True Positive Rate

$$= \frac{\sum TP}{\sum CP}$$

False Positive Rate

$$= \frac{\sum FP}{\sum CN}$$

Accuracy

$$= \frac{\sum TP + \sum TN}{Totalpopulation}$$

G-measure

$$\sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}$$

F1-Score

$$\frac{2\sum TP}{2\sum TP + \sum FN + \sum FP}$$

Positive Predictive Value

$$\frac{\sum TP}{\sum PCP}$$

False Ommision Rate

$$\frac{\sum FN}{\sum PCN}$$

Hamming-Distance Number of changes needed to match the matrices.

References

Sammut, Claude, and Geoffrey I. Webb. (2017). Encyclopedia of machine learning and data mining. Springer.

Examples

```
test.m <- matrix(data = c(0,1,0,
                          0,0,0,
                          1,0,0), nrow = 3, ncol = 3)
ref.m <- matrix(data = c(0,0,0,
                         1,0,0,
                         1,0,0), nrow = 3, ncol = 3)

colnames(test.m) <- rownames(test.m) <- colnames(ref.m) <- rownames(ref.m) <- c("a", "b", "c")

unlist(compareDag(ref = ref.m, test = test.m))
```

`compareEG` *Compare two DAGs or EGs*

Description

Function that returns multiple graph metrics to compare two DAGs or essential graphs, known as confusion matrix or error matrix.

Usage

```
compareEG(ref, test)
```

Arguments

`ref`	a matrix or a formula statement (see details for format) defining the reference network structure, a directed acyclic graph (DAG). Note that row names must be set or given in `node.names` if the DAG is given via a formula statement.
`test`	a matrix or a formula statement (see details for format) defining the test network structure, a directed acyclic graph (DAG). Note that row names must be set or given in `node.names` if the DAG is given via a formula statement.

Details

This R function returns standard Directed Acyclic Graph comparison metrics. In statistical classification, those metrics are known as a confusion matrix or error matrix.

Those metrics allows visualization of the difference between different DAGs. In the case where comparing TRUTH to learned structure or two learned structures, those metrics allow the user to estimate the performance of the learning algorithm. In order to compute the metrics, a contingency table is computed of a pondered difference of the adjacency matrices od the two graphs.

The `ref` or `test` can be provided using a formula statement (similar to GLM input). A typical formula is `~ node1|parent1:parent2 + node2:node3|parent3`. The formula statement have to start with `~`. In this example, node1 has two parents (parent1 and parent2). node2 and node3 have the same parent3. The parents names have to exactly match those given in `node.names`. `:` is the separtor between either children or parents, `|` separates children (left side) and parents (right side), `+` separates terms, `.` replaces all the variables in `node.names`.

To test for essential graphs (or graphs) in general, the test for DAG need to be switched off `checkDAG=FALSE`. The function `compareEG()` is a wrapper to `compareDag(, checkDAG=FALSE)`.

Value

`TP` True Positive

`TN` True Negative

`FP` False Positive

`FN` False Negative

`CP` Condition Positive (ref)

`CN` Condition Negative (ref)

`PCP` Predicted Condition Positive (test)

`PCN` Predicted Condition Negative (test)

`True Positive Rate`

$$= \frac{\sum TP}{\sum CP}$$

`False Positive Rate`

$$= \frac{\sum FP}{\sum CN}$$

`Accuracy`

$$= \frac{\sum TP + \sum TN}{Totalpopulation}$$

`G-measure`

$$\sqrt{\frac{TP}{TP+FP} \cdot \frac{TP}{TP+FN}}$$

`F1-Score`

$$\frac{2\sum TP}{2\sum TP + \sum FN + \sum FP}$$

`Positive Predictive Value`

$$\frac{\sum TP}{\sum PCP}$$

`False Ommision Rate`

$$\frac{\sum FN}{\sum PCN}$$

`Hamming-Distance` Number of changes needed to match the matrices.

References

Sammut, Claude, and Geoffrey I. Webb. (2017). Encyclopedia of machine learning and data mining. Springer.

Examples

```
test.m <- matrix(data = c(0,1,0,
                          0,0,0,
                          1,0,0), nrow = 3, ncol = 3)
ref.m <- matrix(data = c(0,0,0,
                         1,0,0,
                         1,0,0), nrow = 3, ncol = 3)

colnames(test.m) <- rownames(test.m) <- colnames(ref.m) <- rownames(ref.m) <- c("a", "b", "c")

unlist(compareDag(ref = ref.m, test = test.m))
```

`discretization` *Discretization of a Possibly Continuous Data Frame of Random Variables based on their distribution*

Description

This function discretizes a data frame of possibly continuous random variables through rules for discretization. The discretization algorithms are unsupervised and univariate. See details for the complete list of discretization rules (the number of state of each random variable could also be provided).

Usage

```
discretization(data.df = NULL,
                       data.dists = NULL,
                       discretization.method = "sturges",
                       nb.states = FALSE)
```

Arguments

`data.df`	a data frame containing the data to discretize, binary and multinomial variables must be declared as factors, others as a numeric vector. The data frame must be named.
`data.dists`	a named list giving the distribution for each node in the network.
`discretization.method`	a character vector giving the discretization method to use; see details. If a number is provided, the variable will be discretized by equal binning.
`nb.states`	logical variable to select the output. If set to `TRUE` a list with the discretized data frame and the number of state of each variable is returned. If set to FALSE only the discretized data frame is returned.

Details

`fd` Freedman Diaconis rule. `IQR()` stands for interquartile range. The number of bins is given by

$$\frac{range(x) * n^{1/3}}{2 * IQR(x)}$$

The Freedman Diaconis rule is known to be less sensitive than the Scott's rule to outlier.

`doane` Doane's rule. The number of bins is given by

$$1 + \log_2 n + \log_2 1 + \frac{|g|}{\sigma_g}$$

This is a modification of Sturges' formula, which attempts to improve its performance with non-normal data.

`sqrt` The number of bins is given by:

$$\sqrt{(n)}$$

`cencov` Cencov's rule. The number of bins is given by:

$$n^{1/3}$$

`rice` Rice' rule. The number of bins is given by:

$$2n^{1/3}$$

`terrell-scott` Terrell-Scott's rule. The number of bins is given by:

$$(2n)^{1/3}$$

It is known that Cencov, Rice, and Terrell-Scott rules over-estimates k, compared to other rules due to its simplicity.

`sturges` Sturges's rule. The number of bins is given by:

$$1 + \log_2(n)$$

`scott` Scott's rule. The number of bins is given by:

$$range(x)/\sigma(x)n^{-1/3}$$

Value

The discretized data frame or a list containing the table of counts for each bin the discretized data frame.

table of counts for each bin of the discretized data frame.

References

Garcia, S., et al. (2013). A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge and Data Engineering*, 25.4, 734-750.

Cebeci, Z. and Yildiz, F. (2017). Unsupervised Discretization of Continuous Variables in a Chicken Egg Quality Traits Dataset. *Turkish Journal of Agriculture-Food Science and Technology*, 5.4, 315-320.

Examples

```
## Generate random variable
rv <- rnorm(n = 100, mean = 5, sd = 2)
dist <- list("gaussian")
names(dist) <- c("rv")

## Compute the entropy through discretization
entropyData(freqs.table = discretization(data.df = rv, data.dists = dist,
discretization.method = "sturges", nb.states = FALSE))
```

`entropyData` *Computes an Empirical Estimation of the Entropy from a Table of Counts*

Description

This function empirically estimates the Shannon entropy from a table of counts using the observed frequencies.

Usage

```
entropyData(freqs.table)
```

Arguments

`freqs.table`	a table of counts.

Details

The general concept of entropy is defined for probability distributions. The `entropyData()` function estimates empirical entropy from data. The probability is estimated from data using frequency tables. Then the estimates are plug-in in the definition of the entropy to return the so-called empirical entropy. A common known problem of empirical entropy is that the estimations are biased due to the sampling noise. It is also known that the bias will decrease as the sample size increases.

Value

Shannon's entropy estimate on natural logarithm scale.

integer

References

Cover, Thomas M, and Joy A Thomas. (2012). "Elements of Information Theory". John Wiley & Sons.

See Also

`discretization`

Examples

```
## Generate random variable
rv <- rnorm(n = 100, mean = 5, sd = 2)
dist <- list("gaussian")
names(dist) <- c("rv")

## Compute the entropy through discretization
entropyData(freqs.table = discretization(data.df = rv, data.dists = dist,
discretization.method = "sturges", nb.states = FALSE))
```

`essentialGraph` *Construct the essential graph*

Description

Constructs different versions of the essential graph from a given DAG. External function that computes essential graph of a dag Minimal PDAG: The only directed edges are those who participate in v-structure Completed PDAG: very directed edge corresponds to a compelled edge, and every undirected edge corresponds to a reversible edge

Usage

```
essentialGraph(dag, node.names = NULL, PDAG = "minimal")
```

Arguments

`dag`	a matrix or a formula statement (see 'Details' for format) defining the network structure, a directed acyclic graph (DAG).
`node.names`	a vector of names if the DAG is given via formula, see 'Details'.
`PDAG`	a character value that can be: minimal or complete, see 'Details'.

Details

This function returns an essential graph from a DAG, aka acyclic partially directed graph (PDAG). This can be useful if the learning procedure is defined up to a Markov class of equivalence. A minimal PDAG is defined as only directed edges are those who participate in v-structure. Whereas the completed PDAG: every directed edge corresponds to a compelled edge, and every undirected edge corresponds to a reversible edge.

The `dag` can be provided using a formula statement (similar to glm). A typical formula is `~ node1|parent1:parent2 + node2:node3|parent3`. The formula statement have to start with `~`. In this example, node1 has two parents (parent1 and parent2). node2 and node3 have the same parent3. The parents names have to exactly match those given in `node.names`. `:` is the separator between either children or parents, `|` separates children (left side) and parents (right side), `+` separates terms, `.` replaces all the variables in `node.names`.

Value

A matrix giving the PDAG.

References

West, D. B. (2001). Introduction to Graph Theory. Vol. 2. Upper Saddle River: Prentice Hall. Chickering, D. M. (2013) A Transformational Characterization of Equivalent Bayesian Network Structures, arXiv:1302.4938.

Examples

```
dag <- matrix(c(0,0,0, 1,0,0, 1,1,0), nrow = 3, ncol = 3)
dist <- list(a="gaussian", b="gaussian", c="gaussian")
colnames(dag) <- rownames(dag) <- names(dist)

essentialGraph(dag)
```

---

expit *expit of proportions*

Description

See also the C implementation `?abn::expit_cpp()`.

Usage

```
expit(x)
```

Arguments

x	numeric with values between `[0,1]`.

Value

numeric vector of same length as `x`.

---

expit_cpp *expit function*

Description

transform `x` either via the logit, or expit.

Usage

```
expit_cpp(x)
```

Arguments

x	a numeric vector

Value

a numeric vector

`export_abnFit` *Export abnFit object to structured JSON format*

Description

Exports a fitted Additive Bayesian Network (ABN) model to a structured JSON format suitable for storage, sharing, and interoperability with other analysis tools. The export includes network structure (variables and arcs) and model parameters (coefficients, variances, and their associated metadata).

Usage

```
export_abnFit(
  object,
  format = "json",
  include_network = TRUE,
  file = NULL,
  pretty = TRUE,
  scenario_id = NULL,
  label = NULL,
  ...
)
```

Arguments

`object`	An object of class `abnFit`, typically created by `fitAbn`.
`format`	Character string specifying the export format. Currently, only `"json"` is supported.
`include_network`	Logical, whether to include network structure (variables and arcs). Default is `TRUE`.
`file`	Optional character string specifying a file path to save the JSON output. If `NULL` (default), the JSON string is returned.
`pretty`	Logical, whether to format the JSON output with indentation for readability. Default is `TRUE`. Set to `FALSE` for more compact output.
`scenario_id`	Optional character string or numeric identifier for the model run or scenario. Useful for tracking multiple model versions or experiments. Default is `NULL`.
`label`	Optional character string providing a descriptive name or label for the scenario. Default is `NULL`.
`...`	Additional export options (currently unused, reserved for future extensions).

Details

This function provides a standardized way to export fitted ABN models to JSON, facilitating model sharing, archiving, and integration with external tools or databases. The JSON structure is designed to be both human-readable and machine-parseable, following a flat architecture to avoid deep nesting.

Supported Model Types:

The function handles different model fitting methods:

- **MLE without grouping**: Standard maximum likelihood estimation for all supported distributions (Gaussian, Binomial, Poisson, Multinomial). Exports fixed-effect parameters with standard errors.
- **MLE with grouping**: Generalized Linear Mixed Models (GLMM) with group-level random effects. Exports both fixed effects (mu, betas) and random effects (sigma, sigma_alpha).
- **Bayesian**: Placeholder for future implementation of Bayesian model exports including posterior distributions.

JSON Structure Overview:

The exported JSON follows a three-component structure:

- **variables**: An array where each element represents a node/variable in the network with metadata including identifier, attribute name, distribution type, and states (for categorical variables).
- **parameters**: An array where each element represents a model parameter (intercepts, coefficients, variances) with associated values, standard errors, link functions, and parent variable conditions.
- **arcs**: An array where each element represents a directed edge in the network, specifying source and target variable identifiers.

Additionally, optional top-level fields `scenario_id` and `label` can be used to identify and describe the model.

Value

If `file` is `NULL`, returns a character string containing the JSON representation of the model. If `file` is provided, writes the JSON to the specified file and invisibly returns the file path.

JSON Schema

Top-Level Fields:

`scenario_id` Optional string or numeric identifier for the model run. Can be `null`.

`label` Optional descriptive name for the model. Can be `null`.

`variables` Array of variable objects (see Variables section).

`parameters` Array of parameter objects (see Parameters section).

`arcs` Array of arc objects (see Arcs section).

Variables Array:

Each variable object contains:

`variable_id` Unique identifier for the variable (string). This ID is used throughout the JSON to reference this variable in parameters' `source` fields and in arcs' `source_variable_id`/`target_variable_id` fields.

`attribute_name` Original attribute name from the data (string).

`model_type` Distribution type: `"gaussian"`, `"binomial"`, `"poisson"`, or `"multinomial"`.

`states` Array of state objects for multinomial variables only. Each state has `state_id` (used to reference specific categories in parameters), `value_name` (the category label), and `is_baseline` (whether this is the reference category). `NULL` for continuous variables.

Parameters Array:

Each parameter object contains:

`parameter_id` Unique identifier for the parameter (string).

`name` Parameter name (e.g., `"intercept"`, `"prob_2"`, coefficient name, `"sigma"`, `"sigma_alpha"`).

`link_function_name` Link function: `"identity"` (Gaussian), `"logit"` (Binomial, Multinomial), or `"log"` (Poisson).

`source` Object identifying which variable and state this parameter belongs to. Contains `variable_id` (required, references a variable from the variables array) and optional `state_id` (references a specific state for category-specific parameters in multinomial models).

`coefficients` Array of coefficient objects (typically length 1), each with `value`, `stderr` (or `NULL` for mixed models), `condition_type`, and `conditions` array.

Coefficient Condition Types:

- `"intercept"`: Baseline parameter with no parent dependencies
- `"linear_term"`: Effect of a parent variable
- `"CPT_combination"`: Conditional probability table entry (future use)
- `"variance"`: Residual variance (Gaussian/Poisson only)
- `"random_variance"`: Random effect variance (mixed models)
- `"random_covariance"`: Random effect covariance (multinomial mixed models)

Arcs Array:

Each arc object contains:

`source_variable_id` Identifier of the parent/source node.

`target_variable_id` Identifier of the child/target node.

Design Rationale

The JSON structure uses a flat architecture with three parallel arrays rather than deeply nested objects. This design offers several advantages:

- **Database compatibility**: Easy to store in relational or document databases with minimal transformation.
- **Extensibility**: New parameter types or metadata can be added without restructuring existing fields.
- **Parsability**: Simpler to query and transform programmatically.
- **Flexibility**: Supports both CPT-style and GLM(M)-style models through the polymorphic `source` and `conditions` structure.

Parameters are linked to variables through the `source.variable_id` field, with optional `source.state_id` for category-specific parameters in multinomial models. Parent dependencies are encoded in the `conditions` array within each coefficient.

See Also

- `fitAbn` for fitting ABN models
- `buildScoreCache` for structure learning
- `mostProbable` for finding the most probable network structure

Examples

```
## Not run:
# Load example data and fit a model
library(abn)
data(ex1.dag.data)

# Define distributions
mydists <- list(b1 = "binomial", p1 = "poisson", g1 = "gaussian",
                b2 = "binomial", p2 = "poisson", g2 = "gaussian",
                b3 = "binomial", g3 = "gaussian")

# Build score cache
mycache <- buildScoreCache(data.df = ex1.dag.data,
                           data.dists = mydists,
                           method = "mle",
                           max.parents = 2)

# Find most probable DAG
mp_dag <- mostProbable(score.cache = mycache)

# Fit the model
myfit <- fitAbn(object = mp_dag, method = "mle")

# Export to JSON string with metadata
json_export <- export_abnFit(myfit,
                             scenario_id = "example_model_v1",
                             label = "Example ABN Model")

# View the structure
library(jsonlite)
parsed <- fromJSON(json_export)
str(parsed, max.level = 2)

# Export to file
export_abnFit(myfit,
              file = "my_abn_model.json",
              scenario_id = "example_model_v1",
              label = "Example ABN Model",
              pretty = TRUE)

# Export with compact formatting
```

```
compact_json <- export_abnFit(myfit, pretty = FALSE)

# ---
# Mixed-effects model example
# (Requires data with grouping structure)

# Add grouping variable
ex1.dag.data$group <- rep(1:5, length.out = nrow(ex1.dag.data))

# Build cache with grouping
mycache_grouped <- buildScoreCache(data.df = ex1.dag.data,
                                   data.dists = mydists,
                                   method = "mle",
                                   group.var = "group",
                                   max.parents = 2)

# Fit grouped model
myfit_grouped <- fitAbn(object = mp_dag,
                        method = "mle",
                        group.var = "group")

# Export grouped model (includes random effects)
json_grouped <- export_abnFit(myfit_grouped,
                              scenario_id = "grouped_model_v1",
                              label = "Mixed Effects ABN")

## End(Not run)
```

`family.abnFit` *Print family of objects of class* `abnFit`

Description

Print family of objects of class `abnFit`

Usage

```
## S3 method for class 'abnFit'
family(object, ...)
```

Arguments

`object`	Object of class `abnFit`
`...`	additional parameters. Not used at the moment.

Value

prints the distributions for each variable of the fitted model.

`fit.control` *Control the iterations in* `fitAbn`

Description

Allow the user to set restrictions in the `fitAbn` for both the Bayesian and the MLE approach. Control function similar to `build.control`.

Usage

```
fit.control(
  method = "bayes",
  max.mode.error = 10,
  mean = 0,
  prec = 0.001,
  loggam.shape = 1,
  loggam.inv.scale = 5e-05,
  max.iters = 100,
  epsabs = 1e-07,
  error.verbose = FALSE,
  trace = 0L,
  epsabs.inner = 1e-06,
  max.iters.inner = 100,
  finite.step.size = 1e-07,
  hessian.params = c(1e-04, 0.01),
  max.iters.hessian = 10,
  max.hessian.error = 1e-04,
  factor.brent = 100,
  maxiters.hessian.brent = 10,
  num.intervals.brent = 100,
  min.pdf = 0.001,
  n.grid = 250,
  std.area = TRUE,
  marginal.quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975),
  max.grid.iter = 1000,
  marginal.node = NULL,
  marginal.param = NULL,
  variate.vec = NULL,
  ncores = 1,
  cluster.type = "FORK",
  max.irls = 100,
  tol = 1e-11,
  tolPwrss = 1e-07,
  check.rankX = "message+drop.cols",
  check.scaleX = "message+rescale",
  check.conv.grad = "message",
  check.conv.singular = "message",
```

```
  check.conv.hess = "message",
  xtol_abs = 1e-06,
  ftol_abs = 1e-06,
  trace.mblogit = FALSE,
  catcov.mblogit = "free",
  epsilon = 1e-06,
  only_glmmTMB_poisson = FALSE,
  seed = 9062019L
)
```

Arguments

method	a character that takes one of two values: "bayes" or "mle". Overrides method argument from buildScoreCache.
max.mode.error	if the estimated modes from INLA differ by a factor of max.mode.error or more from those computed internally, then results from INLA are replaced by those computed internally. To force INLA always to be used, then max.mode.error=100, to force INLA never to be used max.mod.error=0. See also fitAbn.
mean	the prior mean for all the Gaussian additive terms for each node. INLA argument control.fixed=list(mean.intercept=...) and control.fixed=list(mean=...).
prec	the prior precision ($\tau = \frac{1}{\sigma^2}$) for all the Gaussian additive term for each node. INLA argument control.fixed=list(prec.intercept=...) and control.fixed=list(prec=...).
loggam.shape	the shape parameter in the Gamma distribution prior for the precision in a Gaussian node. INLA argument control.family=list(hyper = list(prec = list(prior="loggamma",pa loggam.inv.scale)))).
loggam.inv.scale	the inverse scale parameter in the Gamma distribution prior for the precision in a Gaussian node. INLA argument control.family=list(hyper = list(prec = list(prior="loggamma",param=c(loggam.shape, loggam.inv.scale)))).
max.iters	total number of iterations allowed when estimating the modes in Laplace approximation. passed to .Call("fit_single_node", ...).
epsabs	absolute error when estimating the modes in Laplace approximation for models with no random effects. Passed to .Call("fit_single_node", ...).
error.verbose	logical, additional output in the case of errors occurring in the optimization. Passed to .Call("fit_single_node", ...).
trace	Non-negative integer. If positive, tracing information on the progress of the "L-BFGS-B" optimization is produced. Higher values may produce more tracing information. (There are six levels of tracing. To understand exactly what these do see the source code.). Passed to .Call("fit_single_node", ...).
epsabs.inner	absolute error in the maximization step in the (nested) Laplace approximation for each random effect term. Passed to .Call("fit_single_node", ...).
max.iters.inner	total number of iterations in the maximization step in the nested Laplace approximation. Passed to .Call("fit_single_node", ...).

`finite.step.size`
: suggested step length used in finite difference estimation of the derivatives for the (outer) Laplace approximation when estimating modes. Passed to `.Call("fit_single_node", ...)`.

`hessian.params`
: a numeric vector giving parameters for the adaptive algorithm, which determines the optimal stepsize in the finite-difference estimation of the hessian. First entry is the initial guess, second entry absolute error. Passed to `.Call("fit_single_node", ...)`.

`max.iters.hessian`
: integer, maximum number of iterations to use when determining an optimal finite difference approximation (Nelder-Mead). Passed to `.Call("fit_single_node", ...)`.

`max.hessian.error`
: if the estimated log marginal likelihood when using an adaptive 5pt finite-difference rule for the Hessian differs by more than `max.hessian.error` from when using an adaptive 3pt rule then continue to minimize the local error by switching to the Brent-Dekker root bracketing method. Passed to `.Call("fit_single_node", ...)`.

`factor.brent`
: if using Brent-Dekker root bracketing method then define the outer most interval end points as the best estimate of h (stepsize) from the Nelder-Mead as $h/factor.brent, h * factor.brent)$. Passed to `.Call("fit_single_node", ...)`.

`maxiters.hessian.brent`
: maximum number of iterations allowed in the Brent-Dekker method. Passed to `.Call("fit_single_node", ...)`.

`num.intervals.brent`
: the number of initial different bracket segments to try in the Brent-Dekker method. Passed to `.Call("fit_single_node", ...)`.

`min.pdf`
: the value of the posterior density function below which we stop the estimation only used when computing marginals, see details.

`n.grid`
: recompute density on an equally spaced grid with `n.grid` points.

`std.area`
: logical, should the area under the estimated posterior density be standardized to exactly one, useful for error checking.

`marginal.quantiles`
: vector giving quantiles at which to compute the posterior marginal distribution at.

`max.grid.iter`
: gives number of grid points to estimate posterior density at when not explicitly specifying a grid used to avoid excessively long computation.

`marginal.node`
: used in conjunction with `marginal.param` to allow bespoke estimate of a marginal density over a specific grid. value from 1 to the number of nodes.

`marginal.param`
: used in conjunction with `marginal.node`. value of 1 is for intercept, see modes entry in results for the appropriate number.

`variate.vec`
: a vector containing the places to evaluate the posterior marginal density, must be supplied if `marginal.node` is not null.

ncores — The number of cores to parallelize to, see 'Details'. If >0, the number of CPU cores to be used. -1 for all available -1 core. Only for `method="mle"`.

cluster.type — The type of cluster to be used, see `?parallel::makeCluster`. abn then defaults to `"PSOCK"` on Windows and `"FORK"` on Unix-like systems. With "FORK" the child process are started with `rscript_args = "--no-environ"` to avoid loading the whole workspace into each child.

max.irls — total number of iterations for estimating network scores using an Iterative Reweighed Least Square algorithm. Is this DEPRECATED?

tol — real number giving the minimal tolerance expected to terminate the Iterative Reweighed Least Square algorithm to estimate network score. Passed to `irls_binomial_cpp_fast_br` and `irls_poisson_cpp_fast`.

tolPwrss — numeric scalar passed to `glmerControl` - the tolerance for declaring convergence in the penalized iteratively weighted residual sum-of-squares step. Similar to `tol`.

check.rankX — character passed to `lmerControl` and `glmerControl` - specifying if `rankMatrix(X)` should be compared with `ncol(X)` and if columns from the design matrix should possibly be dropped to ensure that it has full rank. Defaults to `message+drop.cols`.

check.scaleX — character passed to `lmerControl` and `glmerControl` - check for problematic scaling of columns of fixed-effect model matrix, e.g. parameters measured on very different scales. Defaults to `message+rescale`.

check.conv.grad — character passed to `lmerControl` and `glmerControl` - checking the gradient of the deviance function for convergence. Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

check.conv.singular — character passed to `lmerControl` and `glmerControl` - checking for a singular fit, i.e. one where some parameters are on the boundary of the feasible space (for example, random effects variances equal to 0 or correlations between random effects equal to +/- 1.0). Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

check.conv.hess — character passed to `lmerControl` and `glmerControl` - checking the Hessian of the deviance function for convergence. Defaults to `message` but can be one of "ignore" - skip the test; "warning" - warn if test fails; "message" - print a message if test fails; "stop" - throw an error if test fails.

xtol_abs — Defaults to 1e-6 stop on small change of parameter value. Only for `method='mle'`, `group.var=...`. Default convergence tolerance for fitted `(g)lmer` models is reduced to the value provided here if default values did not fit. This value here is passed to the `optCtrl` argument of `(g)lmer` (see help of `lme4::convergence()`).

ftol_abs — Defaults to 1e-6 stop on small change in deviance. Similar to `xtol_abs`.

trace.mblogit — logical indicating if output should be produced for each iteration. Directly passed to `trace` argument in `mclogit.control`. Is independent of `verbose`.

`catcov.mblogit` Defaults to "free" meaning that there are no restrictions on the covariances of random effects between the logit equations. Set to "diagonal" if random effects pertinent to different categories are uncorrelated or "single" if random effect variances pertinent to all categories are identical.

`epsilon` Defaults to 1e-8. Positive convergence tolerance ϵ that is directly passed to the `control` argument of `mclogit::mblogit` as `mclogit.control`. Only for `method='mle'`, `group.var=...`.

`only_glmmTMB_poisson`
logical, if TRUE only use `glmmTMB` to fit Poisson nodes with random effects. This is useful if `glmer` fails due to convergence issues. Default is FALSE.

`seed` a non-negative integer which sets the seed in `set.seed(seed)`.

Details

Parallelization over all children is possible via the function `foreach` of the package **doParallel**. `ncores=0` or `ncores=1` use single threaded `foreach`. `ncores=-1` uses all available cores but one.

Value

a list of control parameters for the `fitAbn` function.

See Also

`build.control`.

Other fitAbn: `fitAbn()`

Examples

```
ctrlmle <- abn::fit.control(method = "mle",
                        max.irls = 100,
                        tol = 10^-11,
                        tolPwrss = 1e-7,
                        xtol_abs = 1e-6,
                        ftol_abs = 1e-6,
                        epsilon = 1e-6,
                        ncores = 2,
                        cluster.type = "PSOCK",
                        only_glmmTMB_poisson = FALSE,
                        seed = 9062019L)
ctrlbayes <- abn::fit.control(method = "bayes",
                          mean = 0,
                          prec = 0.001,
                          loggam.shape = 1,
                          loggam.inv.scale = 5e-05,
                          max.mode.error = 10,
                          max.iters = 100,
                          epsabs = 1e-07,
                          error.verbose = FALSE,
                          epsabs.inner = 1e-06,
                          max.iters.inner = 100,
```

```
finite.step.size = 1e-07,
hessian.params = c(1e-04, 0.01),
max.iters.hessian = 10,
max.hessian.error = 1e-04,
factor.brent = 100,
maxiters.hessian.brent = 10,
num.intervals.brent = 100,
min.pdf = 0.001,
n.grid = 100,
std.area = TRUE,
marginal.quantiles = c(0.025, 0.25, 0.5, 0.75, 0.975),
max.grid.iter = 1000,
marginal.node = NULL,
marginal.param = NULL,
variate.vec = NULL,
ncores = 1,
cluster.type = NULL,
seed = 9062019L)
```

`getMSEfromModes` *Extract Standard Deviations from all Gaussian Nodes*

Description

Extract Standard Deviations from all Gaussian Nodes

Usage

```
getMSEfromModes(modes, dists)
```

Arguments

`modes`	list of modes.
`dists`	list of distributions.

Value

named numeric vector. Names correspond to node name. Value to standard deviations.

infoDag *Compute standard information for a DAG.*

Description

This function returns standard metrics for DAG description. A list that contains the number of nodes, the number of arcs, the average Markov blanket size, the neighborhood average set size, the parent average set size and children average set size.

Usage

```
infoDag(object, node.names = NULL)
```

Arguments

object	an object of class abnLearned, abnFit. Alternatively, a matrix or a formula statement defining the network structure, a directed acyclic graph (DAG). Note that row names must be set up or given in node.names.
node.names	a vector of names if the DAG is given via formula, see details.

Details

This function returns a named list with the following entries: the number of nodes, the number of arcs, the average Markov blanket size, the neighborhood average set size, the parent average set size, and the children's average set size.

The dag can be provided using a formula statement (similar to glm). A typical formula is ~ node1|parent1:parent2 + node2:node3|parent3. The formula statement have to start with ~. In this example, node1 has two parents (parent1 and parent2). node2 and node3 have the same parent3. The parents names have to exactly match those given in node.names. : is the separator between either children or parents, | separates children (left side) and parents (right side), + separates terms, . replaces all the variables in node.names.

Value

A named list that contains following entries: the number of nodes, the number of arcs, the average Markov blanket size, the neighborhood average set size, the parent average set size and children average set size.

References

West, D. B. (2001). Introduction to graph theory. Vol. 2. Upper Saddle River: Prentice Hall.

Examples

```
## Creating a dag:
dag <- matrix(c(0,0,0,0, 1,0,0,0, 1,1,0,1, 0,1,0,0), nrow = 4, ncol = 4)
dist <- list(a="gaussian", b="gaussian", c="gaussian", d="gaussian")
colnames(dag) <- rownames(dag) <- names(dist)

infoDag(dag)
plot(createAbnDag(dag = dag, data.dists = dist))
```

`linkStrength` *Returns the strengths of the edge connections in a Bayesian Network learned from observational data*

Description

A flexible implementation of multiple proxy for strength measures useful for visualizing the edge connections in a Bayesian Network learned from observational data.

Usage

```
linkStrength(dag,
                    data.df = NULL,
                    data.dists = NULL,
                    method = c("mi.raw",
                               "mi.raw.pc",
                               "mi.corr",
                               "ls",
                               "ls.pc",
                               "stat.dist"),
                    discretization.method = "doane")
```

Arguments

`dag`	a matrix or a formula statement (see details for format) defining the network structure, a directed acyclic graph (DAG). Note that rownames must be set or given in `data.dist` if the DAG is given via a formula statement.
`data.df`	a data frame containing the data used for learning each node, binary variables must be declared as factors.
`data.dists`	a named list giving the distribution for each node in the network, see 'Details'.
`method`	the method to be used. See 'Details'.
`discretization.method`	a character vector giving the discretization method to use. See `discretization`.

Details

This function returns multiple proxies for estimating the connection strength of the edges of a possibly discretized Bayesian network's data set. The returned connection strength measures are: the Raw Mutual Information (`mi.raw`), the Percentage Mutual information (`mi.raw.pc`), the Raw Mutual Information computed via correlation (`mi.corr`), the link strength (`ls`), the percentage link strength (`ls.pc`) and the statistical distance (`stat.dist`).

The general concept of entropy is defined for probability distributions. The probability is estimated from data using frequency tables. Then the estimates are plug-in in the definition of the entropy to return the so-called empirical entropy. A standard known problem of empirical entropy is that the estimations are biased due to the sampling noise. This is also known that the bias will decrease as the sample size increases. The mutual information estimation is computed from the observed frequencies through a plug-in estimator based on entropy. For the case of an arc going from the node X to the node Y and the remaining set of parent of Y is denoted as Z.

The mutual information is defined as I(X, Y) = H(X) + H(Y) - H(X, Y), where H() is the entropy.

The Percentage Mutual information is defined as PI(X,Y) = I(X,Y)/H(Y|Z).

The Mutual Information computed via correlation is defined as MI(X,Y) = -0.5 log(1-cor(X,Y)^2).

The link strength is defined as LS(X->Y) = H(Y|Z)-H(Y|X,Z).

The percentage link strength is defined as PLS(X->Y) = LS(X->Y) / H(Y|Z).

The statistical distance is defined as SD(X,Y) = 1- MI(X,Y) / max(H(X),H(Y)).

Value

The function returns a named matrix with the requested metric.

References

Boerlage, B. (1992). Link strength in Bayesian networks. Diss. University of British Columbia. Ebert-Uphoff, Imme. "Tutorial on how to measure link strengths in discrete Bayesian networks." (2009).

Examples

```
# Gaussian
N <- 1000
mydists <- list(a="gaussian",
                b="gaussian",
                c="gaussian")
a <- rnorm(n = N, mean = 0, sd = 1)
b <- 1 + 2*rnorm(n = N, mean = 5, sd = 1)
c <- 2 + 1*a + 2*b + rnorm(n = N, mean = 2, sd = 1)
mydf <- data.frame("a" = a,
                   "b" = b,
                   "c" = c)
mycache.mle <- buildScoreCache(data.df = mydf,
                               data.dists = mydists,
                               method = "mle",
                               max.parents = 2)
mydag.mp <- mostProbable(score.cache = mycache.mle, verbose = FALSE)
```

```
linkstr <- linkStrength(dag = mydag.mp$dag,
                        data.df = mydf,
                        data.dists = mydists,
                        method = "ls",
                        discretization.method = "sturges")
```

`logit` *Logit of proportions*

Description

See also the C implementation `?abn::logit_cpp()`.

Usage

```
logit(x)
```

Arguments

`x`	numeric with values between `[0,1]`.

Value

numeric vector of same length as `x`.

numeric vector of same length as `x`.

`logit_cpp` *logit functions*

Description

transform `x` either via the logit, or expit.

Usage

```
logit_cpp(x)
```

Arguments

`x`	a numeric vector

Value

a numeric vector

logLik.abnFit *Print logLik of objects of class* abnFit

Description

Print logLik of objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
logLik(object, digits = 3L, verbose = TRUE, ...)
```

Arguments

object	Object of class abnFit
digits	number of digits of the results.
verbose	print additional output.
...	additional parameters. Not used at the moment.

Value

prints the logLik of the fitted model.

mb *Compute the Markov blanket*

Description

This function computes the Markov blanket of a set of nodes given a DAG (Directed Acyclic Graph).

Usage

```
mb(dag, node, data.dists = NULL, data.df = NULL)
```

Arguments

dag	a matrix or a formula statement (see details for format) defining the network structure, a directed acyclic graph (DAG).
node	a character vector of the nodes for which the Markov Blanket should be returned.
data.dists	a named list giving the distribution for each node in the network, see details.
data.df	a data frame containing the data for the nodes in the network. Only needed if dag is a formula statement.

Details

This function returns the Markov Blanket of a set of nodes given a DAG.

The dag can be provided as a matrix where the rows and columns are the nodes names. The matrix should be binary, where 1 indicates an edge from the column node (parent) to the row node (child). The diagonal of the matrix should be 0 and the matrix should be acyclic. The nodes names should be the same as the names of the distributions in `data.dists`.

Alternatively, the dag can be provided using a formula statement (similar to glm). This requires the `data.dists` and `data.df` arguments to be provided. A typical formula is `~ node1|parent1:parent2 + node2:node3|parent3`. The formula statement have to start with `~`. In this example, `node1` has two parents (`parent1` and `parent2`). `node2` and `node3` are children of the same parent (`parent3`). The parents names have to exactly match those given in `name`. `:` is the separator between either children or parents, `|` separates children (left side) and parents (right side), `+` separates terms, `.` replaces all the variables in `name`.

Value

character vector of node names from the Markov blanket.

Examples

```
## Defining distribution and dag
dist <- list(a="gaussian", b="gaussian", c="gaussian", d="gaussian",
             e="binomial", f="binomial")
dag <- matrix(c(0,1,1,0,1,0,
                0,0,1,1,0,1,
                0,0,0,0,0,0,
                0,0,0,0,0,0,
                0,0,0,0,0,1,
                0,0,0,0,0,0), nrow = 6L, ncol = 6L, byrow = TRUE)
colnames(dag) <- rownames(dag) <- names(dist)

mb(dag, node = "b", data.dists = dist)
mb(dag, node = c("b","e"), data.dists = dist)
```

---

miData *Empirical Estimation of the Entropy from a Table of Counts*

Description

This function empirically estimates the Mutual Information from a table of counts using the observed frequencies.

Usage

```
miData(freqs.table, method = c("mi.raw", "mi.raw.pc"))
```

Arguments

freqs.table	a table of counts.
method	a character determining if the Mutual Information should be normalized.

Details

The mutual information estimation is computed from the observed frequencies through a plugin estimator based on entropy.

The plugin estimator is

$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

, where

$$H()$$

is the entropy computed with entropyData.

Value

Mutual information estimate.

integer

References

Cover, Thomas M, and Joy A Thomas. (2012). "Elements of Information Theory". John Wiley & Sons.

See Also

discretization

Examples

```
## Generate random variable
Y <- rnorm(n = 100, mean = 0, sd = 2)
X <- rnorm(n = 100, mean = 5, sd = 2)

dist <- list(Y="gaussian", X="gaussian")

miData(discretization(data.df = cbind(X,Y), data.dists = dist,
                      discretization.method = "fd", nb.states = FALSE),
                      method = "mi.raw")
```

modes2coefs *Convert modes to fitAbn.mle$coefs structure*

Description

Convert modes to fitAbn.mle$coefs structure

Usage

```
modes2coefs(modes)
```

Arguments

modes	list of modes.

Value

list of matrix arrays.

mostProbable *Find most probable DAG structure*

Description

Find most probable DAG structure using exact order based approach due to Koivisto and Sood, 2004.

Usage

```
mostProbable(score.cache, score="bic", prior.choice=1, verbose=TRUE, ...)
```

Arguments

score.cache	object of class abnCache typically outputted by from buildScoreCache().
score	which score should be used to score the network. Possible choices are aic, bic, mdl, mlik.
prior.choice	an integer, 1 or 2, where 1 is a uniform structural prior and 2 uses a weighted prior, see details
verbose	if TRUE then provides some additional output.
...	further arguments passed to or from other methods.

Details

The procedure runs the exact order based structure discovery approach of Koivisto and Sood (2004) to find the most probable posterior network (DAG). The local.score is the node cache, as created using buildScoreCache (or manually provided the same format is used). Note that the scope of this search is given by the options used in local.score, for example, by restricting the number of parents or the ban or retain constraints given there.

This routine can take a long time to complete and is highly sensitive to the number of nodes in the network. It is recommended to use this on a reduced data set to get an idea as to the computational practicality of this approach. In particular, memory usage can quickly increase to beyond what may be available. For additive models, problems comprising up to 20 nodes are feasible on most machines. Memory requirements can increase considerably after this, but then so does the run time making this less practical. It is recommended that some form of over-modeling adjustment is performed on this resulting DAG (unless dealing with vast numbers of observations), for example, using parametric bootstrapping, which is straightforward to implement in MCMC engines such as JAGS or WinBUGS. See the case studies at https://r-bayesian-networks.org/ or the files provided in the package directories inst/bootstrapping_example and inst/old_vignette for details.

The parameter prior.choice determines the prior used within each node for a given choice of parent combination. In Koivisto and Sood (2004) p.554, a form of prior is used, which assumes that the prior probability for parent combinations comprising of the same number of parents are all equal. Specifically, that the prior probability for parent set G with cardinality |G| is proportional to 1/[n-1 choose |G|] where there are n total nodes. Note that this favors parent combinations with either very low or very high cardinality, which may not be appropriate. This prior is used when prior.choice=2. When prior.choice=1 an uninformative prior is used where parent combinations of all cardinalities are equally likely. The latter is equivalent to the structural prior used in the heuristic searches e.g., searchHillclimber or searchHeuristic.

Note that the network score (log marginal likelihood) of the most probable DAG is not returned as it can easily be computed using fitAbn, see examples below.

Value

An object of class abnMostprobable, which is a list containing: a matrix giving the DAG definition of the most probable posterior structure, the cache of pre-computed scores and the score used for selection.

References

Koivisto, M. V. (2004). Exact Structure Discovery in Bayesian Networks, Journal of Machine Learning Research, vol 5, 549-573.

Examples

```
## Not run:
##############################
## Example 1
##############################
## This data comes with 'abn' see ?ex1.dag.data
mydat <- ex1.dag.data[1:5000, c(1:7, 10)]
```

```
## Setup distribution list for each node:
mydists <- list(b1 = "binomial",
                p1 = "poisson",
                g1 = "gaussian",
                b2 = "binomial",
                p2 = "poisson",
                b3 = "binomial",
                g2 = "gaussian",
                g3 = "gaussian")

## Parent limits, for speed purposes quite specific here:
max_par <- list("b1" = 0,
                "p1" = 0,
                "g1" = 1,
                "b2" = 1,
                "p2" = 2,
                "b3" = 3,
                "g2" = 3,
                "g3" = 2)
## Now build cache (no constraints in ban nor retain)
mycache <- buildScoreCache(data.df = mydat,
                           data.dists = mydists,
                           max.parents = max_par)

## Find the globally best DAG:
mp_dag <- mostProbable(score.cache = mycache)
myres <- fitAbn(object = mp_dag,
                create.graph = TRUE)
plot(myres) # plot the best model

## Fit the known true DAG (up to variables 'b4' and 'b5'):
true_dag <- matrix(data = 0, ncol = 8, nrow = 8)
colnames(true_dag) <- rownames(true_dag) <- names(mydists)

true_dag["p2", c("b1", "p1")] <- 1
true_dag["b3", c("b1", "g1", "b2")] <- 1
true_dag["g2", c("p1", "g1", "b2")] <- 1
true_dag["g3", c("g1", "b2")] <- 1

fitAbn(dag = true_dag,
       data.df = mydat,
       data.dists = mydists)$mlik

#####################################################################
## Example 2 - models with random effects
#####################################################################
## This data comes with abn see ?ex3.dag.data
mydat <- ex3.dag.data[, c(1:4, 14)]
mydists <- list(b1 = "binomial",
                b2 = "binomial",
                b3 = "binomial",
                b4 = "binomial")
```

```
## This takes a few seconds and requires INLA:
mycache_mixed <- buildScoreCache(data.df = mydat,
                                 data.dists = mydists,
                                 group.var = "group",
                                 max.parents = 2)

## Find the most probable DAG:
mp_dag <- mostProbable(score.cache = mycache_mixed)
## and get goodness of fit:
fitAbn(object = mp_dag,
       group.var = "group")$mlik

## End(Not run)
```

---

nobs.abnFit *Print number of observations of objects of class* abnFit

---

Description

Print number of observations of objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
nobs(object, ...)
```

Arguments

object	Object of class abnFit
...	additional parameters. Not used at the moment.

Value

prints the number of observations of the fitted model.

---

odds *Probability to odds*

Description

Probability to odds

Usage

```
odds(x)
```

Arguments

x	numeric vector of probabilities with values between [0,1].

Value

numeric vector of same length as x.

---

or *Odds Ratio from a matrix*

Description

Compute the odds ratio from a contingency table or a matrix.

Usage

```
or(x)
```

Arguments

x	a 2x2 table or matrix.

Value

A real value.

---

plot.abnDag *Plots DAG from an object of class* abnDag

Description

Plots DAG from an object of class abnDag

Usage

```
## S3 method for class 'abnDag'
plot(x, ...)
```

Arguments

x	Object of class abnDag
...	additional parameters. Not used at the moment.

Value

Rgraphviz::plot

Examples

```
mydag <- createAbnDag(dag = ~a+b|a,
                      data.df = data.frame("a"=1, "b"=1),
                      data.dists = list(a="binomial", b="gaussian"))
plot(mydag)
```

---

plot.abnFit *Plot objects of class* abnFit

Description

Plot objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
plot(x, ...)
```

Arguments

x	Object of class abnFit
...	additional parameters. Not used at the moment.

Value

a plot of the fitted model.

---

plot.abnHeuristic *Plot objects of class* abnHeuristic

Description

Plot objects of class abnHeuristic

Usage

```
## S3 method for class 'abnHeuristic'
plot(x, ...)
```

Arguments

x	Object of class abnHeuristic
...	additional parameters. Not used at the moment.

Value

plot of the scores of the heuristic search.

plot.abnHillClimber *Plot objects of class* abnHillClimber

Description

Plot objects of class abnHillClimber

Usage

```
## S3 method for class 'abnHillClimber'
plot(x, ...)
```

Arguments

x	Object of class abnHillClimber
...	additional parameters. Not used at the moment.

Value

plot of the consensus DAG.

plot.abnMostprobable *Plot objects of class* abnMostprobable

Description

Plot objects of class abnMostprobable

Usage

```
## S3 method for class 'abnMostprobable'
plot(x, ...)
```

Arguments

x	Object of class abnMostprobable
...	additional parameters. Not used at the moment.

Value

plot of the mostprobable consensus DAG.

print.abnCache *Print objects of class* abnCache

Description

Print objects of class abnCache

Usage

```
## S3 method for class 'abnCache'
print(x, digits = 3, ...)
```

Arguments

x	Object of class abnCache
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

summary statement of the class of abnCache.

Examples

```
## Subset of the build-in dataset, see  ?ex0.dag.data
mydat <- ex0.dag.data[,c("b1","b2","g1","g2","b3","g3")] ## take a subset of cols

## setup distribution list for each node
mydists <- list(b1="binomial", b2="binomial", g1="gaussian",
                g2="gaussian", b3="binomial", g3="gaussian")

# Structural constraints
# ban arc from b2 to b1
# always retain arc from g2 to g1

## parent limits
max.par <- list("b1"=2, "b2"=2, "g1"=2, "g2"=2, "b3"=2, "g3"=2)

## now build the cache of pre-computed scores accordingly to the structural constraints
if(requireNamespace("INLA", quietly = TRUE)){
  # Run only if INLA is available
res.c <- buildScoreCache(data.df=mydat, data.dists=mydists,
                         dag.banned= ~b1|b2, dag.retained= ~g1|g2, max.parents=max.par)
print(res.c)
}
```

print.abnDag *Print objects of class* abnDag

Description

Print objects of class abnDag

Usage

```
## S3 method for class 'abnDag'
print(x, digits = 3L, ...)
```

Arguments

x	Object of class abnDag
digits	number of digits of the adjacency matrix.
...	additional parameters. Not used at the moment.

Value

outputs adjacency matrix and statement of the class of x.

Examples

```
mydag <- createAbnDag(dag = ~a+b|a, data.df = data.frame("a"=1, "b"=1))
print(mydag)
```

print.abnFit *Print objects of class* abnFit

Description

Print objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
print(x, digits = 3L, ...)
```

Arguments

x	Object of class abnFit
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

prints the parameters of the fitted model.

---

print.abnHeuristic *Print objects of class* abnHeuristic

Description

Print objects of class abnHeuristic

Usage

```
## S3 method for class 'abnHeuristic'
print(x, digits = 2L, ...)
```

Arguments

x	Object of class abnHeuristic
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

prints the best score found and the distribution of the scores.

---

print.abnHillClimber *Print objects of class* abnHillClimber

Description

Print objects of class abnHillClimber

Usage

```
## S3 method for class 'abnHillClimber'
print(x, digits = 3L, ...)
```

Arguments

x	Object of class abnHillClimber
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

prints the consensus DAG and the class of the object.

print.abnMostprobable *Print objects of class* abnMostprobable

Description

Print objects of class abnMostprobable

Usage

```
## S3 method for class 'abnMostprobable'
print(x, digits = 3L, ...)
```

Arguments

x	Object of class abnMostprobable
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

prints the mostprobable consensus DAG.

scoreContribution *Compute the score's contribution in a network of each observation.*

Description

This function computes the score's contribution of each observation to the total network score.

Usage

```
scoreContribution(object = NULL,
                          dag = NULL, data.df = NULL, data.dists = NULL,
                          verbose = FALSE)
```

Arguments

object	an object of class 'abnLearned' produced by mostProbable, searchHeuristic or searchHillClimber.
dag	a matrix or a formula statement (see details) defining the network structure, a directed acyclic graph (DAG), see details for format. Note that colnames and rownames must be set.
data.df	a data frame containing the data used for learning the network, binary variables must be declared as factors and no missing values all allowed in any variable.
data.dists	a named list giving the distribution for each node in the network, see details.
verbose	if TRUE then provides some additional output.

Details

This function computes the score contribution of each observation to the total network score. This function is available only in the `mle` settings. To do so one uses the `glm` and `predict` functions. This function is an attempt to perform diagnostic for an ABN analysis.

Value

A named list that contains the scores contributions: maximum likelihood, aic, bic, mdl and diagonal values of the hat matrix.

Examples

```
## Not run:
## Use a subset of a built-in simulated data set
mydat <- ex1.dag.data[,c("b1","g1","p1")]

## setup distribution list for each node
mydists <- list(b1="binomial", g1="gaussian", p1="poisson")

## now build cache
mycache <- buildScoreCache(data.df = mydat, data.dists = mydists, max.parents = 2, method = "mle")

## Find the globally best DAG
mp.dag <- mostProbable(score.cache=mycache, score="bic", verbose = FALSE)

out <- scoreContribution(object = mp.dag)

## Observations contribution per network node
boxplot(out$bic)

## End(Not run)
```

---

searchHeuristic *A family of heuristic algorithms that aims at finding high scoring directed acyclic graphs*

Description

A flexible implementation of multiple greedy search algorithms to find high scoring network (DAG)

Usage

```
searchHeuristic(score.cache, score = "mlik",
                        num.searches = 1, seed = 42L, start.dag = NULL,
                        max.steps = 100,
                        algo = "hc", tabu.memory = 10, temperature = 0.9,
                        verbose = FALSE, ...)
```

Arguments

score.cache	output from buildScoreCache().
score	which score should be used to score the network. Possible choices are aic, bic, mdl, mlik.
num.searches	a positive integer giving the number of different search to run, see details.
seed	a non-negative integer which sets the seed.
start.dag	a DAG given as a matrix, see details for format, which can be used to explicity provide a starting point for the structural search.
max.steps	a constant giving the number of search steps per search, see details.
algo	which heuristic algorithm should be used. Possible choices are: hc, tabu, sa.
tabu.memory	a non-negative integer number to set the memory of the tabu search.
temperature	a real number giving the update in temperature for the sa (simulated annealing) search algorithm.
verbose	if TRUE then provides some additional output.
...	further arguments passed to or from other methods.

Details

This function is a flexible implementation of multiple greedy heuristic algorithms, particularly well adapted to the abn framework. It targets multi-random restarts heuristic algorithms. The user can select the num.searches and the maximum number of steps within by max.steps. The optimization algorithm within each search is relatively rudimentary.

The function searchHeuristic is different from the searchHillClimber in the sense that this function is fully written in R, whereas the searchHillClimber is written in C and thus expected to be faster. The function searchHillClimber at each hill-climbing step consider every information from the pre-computed scores cache while the function searchHeuristic performs a local stepwise optimization. This function chooses a structural move (or edge move) and compute the score's change. On this point, it is closer to the MCMCMC algorithm from Madigan and York (1995) and Giudici and Castelo (2003) with a single edge move.

If the user select random, then a random valid DAG is selected. The routine used favourise low density structure. The function implements three algorithm selected with the parameter algo: hc, tabu or sa.

If algo=hc: The Hill-climber algorithm (hc) is a single move algorithm. At each Hill-climbing step within a search an arc is attempted to be added. The new score is computed and compared to the previous network's score.

If algo=tabu: The same algorithm is as with hc is used, but a list of banned moves is computed. The parameter tabu.memory controls the length of the tabu list. The idea is that the classical Hill-climber algorithm is inefficient when it should cross low probability regions to unblock from a local maximum and reaching a higher score peak. By forcing the algorithm to choose some not already used moves, this will force the algorithm to escape the local maximum.

If algo=sa: This variant of the heuristic search algorithm is based on simulated annealing described by Metropolis et al. (1953). Some accepted moves could result in a decrease of the network score. The acceptance rate can be monitored with the parameter temperature.

Value

An object of class abnHeuristic (which extends the class abnLearnd) and contains list with entires:

dags a list of DAGs

scores a vector giving the network score for the locally optimal network for each search

detailed.score a vector giving the evolution of the network score for the all the random restarts

score a number giving the network score for the locally optimal network

score.cache the pre-computed cache of scores

num.searches a numeric giving the number of random restart

max.steps a numeric giving the maximal number of optimization steps within each search

algorithm a character for indicating the algorithm used

References

Heckerman, D., Geiger, D. and Chickering, D. M. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20, 197-243. Madigan, D. and York, J. (1995) "Bayesian graphical models for discrete data". International Statistical Review, 63:215232. Giudici, P. and Castelo, R. (2003). "Improving Markov chain Monte Carlo model search for data mining". Machine Learning, 50:127158. Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). "Equation of state calculations by fast computing machines". The journal of chemical physics, 21(6), 1087-1092.

Examples

```
## Not run:
##############################################
## example: use built-in simulated data set
##############################################

mydat <- ex1.dag.data ## this data comes with abn see ?ex1.dag.data

## setup distribution list for each node
mydists<-list(b1="binomial", p1="poisson", g1="gaussian", b2="binomial",
              p2="poisson", b3="binomial", g2="gaussian", b4="binomial",
              b5="binomial", g3="gaussian")

mycache <- buildScoreCache(data.df = mydat, data.dists = mydists, max.parents = 2)

## Now peform 10 greedy searches
heur.res <- searchHeuristic(score.cache = mycache, data.dists = mydists,
                            start.dag = "random", num.searches = 10,
                            max.steps = 50)

## Plot (one) dag
plotAbn(heur.res$dags[[1]], data.dists = mydists)

## End(Not run)
```

`searchHillClimber` *Find high scoring directed acyclic graphs using heuristic search.*

Description

Find high scoring network (DAG) structures using a random re-starts greedy hill-climber heuristic search.

Usage

```
searchHillClimber(score.cache, score = "mlik", num.searches = 1, seed = 42,
                    start.dag = NULL, support.threshold = 0.5, timing.on = TRUE,
                         dag.retained = NULL, verbose = FALSE, ...)
```

Arguments

`score.cache`	output from `buildScoreCache()`.
`score`	character giving which network score should be used to select the structure. Currently `'mlik'` only.
`num.searches`	number of times to run the search.
`seed`	non-negative integer which sets the seed in the GSL random number generator.
`start.dag`	a DAG given as a matrix, see details for format, which can be used to provide a starting point for the structural search explicitly.
`support.threshold`	the proportion of search results - each locally optimal DAG - in which each arc must appear to be a part of the consensus network.
`timing.on`	extra output in terms of duration computation.
`dag.retained`	a DAG given as a matrix, see details for format. This is necessary if the score.cache was created using an explicit retain matrix, and the same retain matrix should be used here. dag.retained is used by the algorithm which generates the initial random DAG to ensure that the necessary arcs are retained.
`verbose`	extra output.
`...`	further arguments passed to or from other methods.

Details

The procedure runs a greedy hill-climbing search similar, but not identical, to the method presented initially in Heckerman et al. 1995. (Machine Learning, 20, 197-243). Each search begins with a randomly chosen DAG structure where this is constructed in such a way as to attempt to choose a DAG uniformly from the vast landscape of possible structures. The algorithm used is as follows: given a node cache (from `buildScoreCache`, then within this set of all allowed local parent combinations, a random combination is chosen for each node. This is then combined into a full DAG, which is then checked for cycles, where this check iterates over the nodes in a random order. If all nodes have at least one child (i.e., at least one cycle is present), then the first node examined has all

its children removed, and the check for cycles is then repeated. If this has removed the only cycle present, then this DAG is used at the starting point for the search, if not, a second node is chosen (randomly) and the process is then repeated until a DAG is obtained.

The actual hill-climbing algorithm itself differs slightly from that presented in Heckerman et al. as a full cache of all possible local combinations are available. At each hill-climbing step, everything in the node cache is considered. In other words, all possible single swaps between members of cache currently present in the DAG and those in the full cache. The single swap, which provides the greatest increase in goodness of fit is chosen. A single swap here is the removal or addition of any one node-parent combination present in the cache while avoiding a cycle. This means that as well as all single arc changes (addition or removal), multiple arc changes are also considered at each same step, note however that arc reversal in this scheme takes two steps (as this requires first removal of a parent arc from one node and then addition of a parent arc to a different node). The original algorithm perturbed the current DAG by only a single arc at each step but also included arc reversal. The current implementation may not be any more efficient than the original but is arguably more natural given a pre-computed cache of local scores.

A start DAG may be provided in which case num.searches must equal 1 - this option is really just to provide a local search around a previously identified optimal DAG.

This function is designed for two different purposes: i) interactive visualization; and ii) longer batch processing. The former provides easy visual "eyeballing" of data in terms of its majority consensus network (or similar threshold), which is a graphical structure which comprises of all arcs which feature in a given proportion (`support.threshold`) of locally optimal DAGs already identified during the run. The general hope is that this structure will stabilize - become fixed - relatively quickly, at least for problems with smaller numbers of nodes.

Value

A list with entries:

init.score a vector giving network score for initial network from which the search commenced

final.score a vector giving the network score for the locally optimal network

init.dag list of matrices, initial DAGs

final.dag list of matrices, locally optimal DAGs

consensus a matrix holding a binary graph, not necessary a DAG!

support.threshold percentage supported used to create consensus matrix

References

Lewis, F. I., and McCormick, B. J. J. (2012). Revealing the complexity of health determinants in resource poor settings. *American Journal Of Epidemiology*. DOI:10.1093/aje/KWS183).

simulateAbn *Simulate data from a fitted additive Bayesian network.*

Description

Simulate data from a fitted additive Bayesian network.

Usage

```
simulateAbn(
  object = NULL,
  run.simulation = TRUE,
  bugsfile = NULL,
  n.chains = 10L,
  n.adapt = 1000L,
  n.thin = 100L,
  n.iter = 10000L,
  seed = 42L,
  verbose = FALSE,
  debug = FALSE
)
```

Arguments

object	of type abnFit.
run.simulation	call JAGS to simulate data (default is TRUE).
bugsfile	A path to a valid file or NULL (default) to delete the bugs file after simulation.
n.chains	number of parallel chains for the model.
n.adapt	number of iteration for adaptation. If n.adapt is set to zero, then no adaptation takes place.
n.thin	thinning interval for monitors.
n.iter	number of iteration to monitor.
seed	by default set to 42.
verbose	if TRUE prints additional output
debug	if TRUE prints bug file content to stdout and does not run simulations.

Value

data.frame

See Also

makebugs

Examples

```
df <- FCV[, c(12:15)]
mydists <- list(Outdoor="binomial",
                Sex="multinomial",
                GroupSize="poisson",
                Age="gaussian")

## buildScoreCache -> mostProbable() -> fitAbn()
suppressWarnings({
  mycache.mle <- buildScoreCache(data.df = df, data.dists = mydists, method = "mle",
                                 adj.vars = NULL, cor.vars = NULL,
                                 dag.banned = NULL, dag.retained = NULL,
                                 max.parents = 1,
                                 which.nodes = NULL, defn.res = NULL)
}) # ignore non-convergence warnings
mp.dag.mle <- mostProbable(score.cache = mycache.mle, verbose = FALSE)
myres.mle <- fitAbn(object = mp.dag.mle, method = "mle")

myres.sim <- simulateAbn(object = myres.mle,
                             run.simulation = TRUE,
                             bugsfile = NULL,
                             verbose = FALSE)
str(myres.sim)
prop.table(table(myres.sim$Outdoor))
prop.table(table(df$Outdoor))
```

`simulateDag` *Simulate a DAG with with arbitrary arcs density*

Description

Provided with node names, returns an abnDAG. Arc density refers to the chance of a node being connected to the node before it.

Usage

```
simulateDag(node.name, data.dists = NULL, edge.density = 0.5, verbose = FALSE)
```

Arguments

node.name	a vector of character giving the names of the nodes. It gives the size of the simulated DAG.
data.dists	named list corresponding to the node.name specifying the distribution for each node. If not provided arbitrary distributions are assigned to the nodes.
edge.density	number in [0,1] specifying the edge probability in the dag.
verbose	print more information on the run.

Details

This function generates DAGs by sampling triangular matrices and reorder columns and rows randomly. The network density (edge.density) is used column-wise as binomial sampling probability. Then the matrix is named using the user-provided names.

Value

object of class abnDag consisting of a named matrix, a named list giving the distribution for each node and an empty element for the data.

Examples

```
simdag <- simulateDag(node.name = c("a", "b", "c", "d"),
                      edge.density = 0.5,
                      data.dists = list(a = "gaussian",
                                        b = "binomial",
                                        c = "poisson",
                                        d = "multinomial"))

## Example using Ozon entries:
dist <- list(Ozone="gaussian",   Solar.R="gaussian",  Wind="gaussian",
             Temp="gaussian",    Month="gaussian",    Day="gaussian")
out <- simulateDag(node.name = names(dist), data.dists = dist, edge.density = 0.8)
plot(out)
```

---

skewness *Computes skewness of a distribution*

Description

Computes skewness of a distribution

Usage

```
skewness(x)
```

Arguments

x	a numeric vector

Value

integer

summary.abnDag *Prints summary statistics from an object of class* abnDag

Description

Prints summary statistics from an object of class abnDag

Usage

```
## S3 method for class 'abnDag'
summary(object, ...)
```

Arguments

object	an object of class abnLearned, abnFit. Alternatively, a matrix or a formula statement defining the network structure, a directed acyclic graph (DAG). Note that row names must be set up or given in node.names.
...	additional parameters. Not used at the moment.

Value

List with summary statistics of the DAG.

Examples

```
mydag <- createAbnDag(dag = ~a+b|a, data.df = data.frame("a"=1, "b"=1))
summary(mydag)
```

summary.abnFit *Print summary of objects of class* abnFit

Description

Print summary of objects of class abnFit

Usage

```
## S3 method for class 'abnFit'
summary(object, digits = 3L, ...)
```

Arguments

object	Object of class abnFit
digits	number of digits of the results.
...	additional parameters. Not used at the moment.

Value

prints summary statistics of the fitted model.

---

summary.abnMostprobable *Print summary of objects of class* abnMostprobable

---

Description

Print summary of objects of class abnMostprobable

Usage

```
## S3 method for class 'abnMostprobable'
summary(object, ...)
```

Arguments

object	Object of class abnMostprobable
...	additional parameters. Not used at the moment.

Value

prints the mostprobable consensus DAG and the number of observations used to calculate it.

---

toGraphviz *Convert a DAG into graphviz format*

Description

Given a matrix defining a DAG create a text file suitable for plotting with graphviz.

Usage

```
toGraphviz(dag,
                  data.df=NULL,
                  data.dists=NULL,
                  group.var=NULL,
                  outfile=NULL,
                  directed=TRUE,
                  verbose=FALSE)
```

Arguments

dag	a matrix defining a DAG.
data.df	a data frame containing the data used for learning the network.
data.dists	a list with named arguments matching the names of the data frame which gives the distribution family for each variable. See fitAbn for details.
group.var	only applicable for mixed models and gives the column name in data.df of the grouping variable (which must be a factor denoting group membership). See fitAbn for details.
outfile	a character string giving the filename which will contain the graphviz graph.
directed	logical; if TRUE, a directed acyclic graph is produced, otherwise an undirected graph.
verbose	if TRUE more output is printed. If TRUE and 'outfile=NULL' the '.dot' file is printed to console.

Details

Graphviz (https://www.graphviz.org) is a visualisation software developed by AT&T and freely available. This function creates a text representation of the DAG, or the undirected graph, so this can be plotted using graphviz. The R package, Rgraphviz (available through the Bioconductor project https://www.bioconductor.org/) interfaces R and the working installation of graphviz.

Binary nodes will appear as squares, Gaussian as ovals and Poisson nodes as diamonds in the resulting graphviz network diagram. There are many other shapes possible for nodes and numerous other visual enhancements - see online graphviz documentation.

Bespoke refinements can be added by editing the raw outfile produced. For full manual editing, particularly of the layout, or adding annotations, one easy solution is to convert a postscript format graph (produced in graphviz using the -Tps switch) into a vector format using a tool such as pstoedit (http://www.pstoedit.net/), and then edit using a vector drawing tool like xfig. This can then be resaved as postscript or pdf thus retaining full vector quality.

Value

Nothing is returned, but a file outfile written.

Author(s)

Fraser Iain Lewis

Marta Pittavino

Examples

```
## On a typical linux system the following code constructs a nice
## looking pdf file 'graph.pdf'.
## Not run:
## Subset of a build-in dataset
mydat <- ex0.dag.data[,c("b1","b2","b3","g1","b4","p2","p4")]

## setup distribution list for each node
```

```
mydists <- list(b1="binomial", b2="binomial", b3="binomial",
                g1="gaussian", b4="binomial", p2="poisson",
                               p4="poisson")
## specify DAG model
mydag <- matrix(c(   0,1,0,0,1,0,0, #
                     0,0,0,0,0,0,0, #
                     0,1,0,0,1,0,0, #
                     1,0,0,0,0,0,1, #
                     0,0,0,0,0,0,0, #
                     0,0,0,1,0,0,0, #
                     0,0,0,0,1,0,0  #
), byrow=TRUE, ncol=7)

colnames(mydag) <- rownames(mydag) <- names(mydat)

## create file for processing with graphviz
outfile <- paste(tempdir(), "graph.dot", sep="/")
toGraphviz(dag=mydag, data.df=mydat, data.dists=mydists, outfile=outfile)
## and then process using graphviz tools e.g. on linux
if(Sys.info()[["sysname"]] == "Linux" && interactive()) {
  system(paste( "dot -Tpdf -o graph.pdf", outfile))
  system("evince graph.pdf")
}
## Example using data with a group variable  where b1<-b2
mydag <- matrix(c(0,1, 0,0), byrow=TRUE, ncol=2)

colnames(mydag) <- rownames(mydag) <- names(ex3.dag.data[,c(1,2)])
## specific distributions
mydists <- list(b1="binomial", b2="binomial")

## create file for processing with graphviz
outfile <- paste0(tempdir(), "/graph.dot")
toGraphviz(dag=mydag, data.df=ex3.dag.data[,c(1,2,14)], data.dists=mydists,
           group.var="group",
           outfile=outfile, directed=FALSE)
## and then process using graphviz tools e.g. on linux:
if(Sys.info()[["sysname"]] == "Linux" && interactive()) {
  pdffile <- paste0(tempdir(), "/graph.pdf")
  system(paste("dot -Tpdf -o ", pdffile, outfile))
  system(paste("evince ", pdffile, " &"))   ## or some other viewer
}

## End(Not run)
```

Index