Package ‘bnlearn’

September 27, 2012

Type Package

Title Bayesian network structure learning, parameter learning and inference

Version 3.1

Date 2012-09-25

Depends R (>= 2.13.2), methods, graph

Suggests snow, Rgraphviz, lattice, gRain

Author Marco Scutari

Maintainer Marco Scutari <marco.scutari@gmail.com>

Description Bayesian network structure learning (via constraint-based,score-based and hybrid algorithms), parameter learning (via ML and Bayesian estimators) and inference. This package implements the Grow-Shrink (GS) algorithm, the Incremental Association (IAMB) algorithm, the Interleaved-IAMB (Inter-IAMB) algorithm, the Fast-IAMB (Fast-IAMB) algorithm, the Max-Min Parents and Children (MMPC) algorithm, the Hiton-PC algorithm,the ARACNE and Chow-Liu algorithms, the Hill-Climbing (HC) greedy search algorithm, the Tabu Search (TABU) algorithm, the Max-Min Hill-Climbing (MMHC) algorithm and the two-stage Restricted Maximization (RSMAX2) algorithm for both discrete and Gaussian networks, along with many score functions and conditional independence tests. The Naive Bayes and the Tree-Augmented Naive Bayes (TAN) classifiers are also implemented. Some utility functions (model comparison and manipulation, random data generation, arc orientation testing,simple and advanced plots) are included, as well as support for parameter estimation and inference, conditional probability queries and cross-validation.

URL http://www.bnlearn.com/

License GPL (>= 2)

LazyLoad yes

LazyData yes

Repository CRAN

Date/Publication 2012-09-27 10:03:49

R topics documented:

`bnlearn-package` *Bayesian network structure learning, parameter learning and inference.*

Description

Bayesian network structure learning (via constraint-based, score-based and hybrid algorithms), parameter learning (via ML and Bayesian estimators) and inference.

Details

Package:	bnlearn
Type:	Package
Version:	3.1
Date:	2012-09-25
License:	GPLv2 or later

This package implements some algorithms for learning the structure of Bayesian networks.

Constraint-based algorithms, also known as *conditional independence learners*, are all optimized derivatives of the *Inductive Causation* algorithm (Verma and Pearl, 1991). These algorithms use conditional independence tests to detect the Markov blankets of the variables, which in turn are used to compute the structure of the Bayesian network.

Score-based learning algorithms are general purpose heuristic optimization algorithms which rank network structures with respect to a goodness-of-fit score.

Hybrid algorithms combine aspects of both constraint-based and score-based algorithms, as they use conditional independence tests (usually to reduce the search space) and network scores (to find the optimal network in the reduced space) at the same time.

Several functions for parameter estimation, parametric inference, bootstrap, cross-validation and stochastic simulation are available. Furthermore, advanced plotting capabilities are implemented on top of the **Rgraphviz** and **lattice** packages.

Available constraint-based learning algorithms

- *Grow-Shrink* (`gs`): based on the *Grow-Shrink Markov Blanket*, the first (and simplest) Markov blanket detection algorithm (Margaritis, 2003) used in a structure learning algorithm.

- *Incremental Association* (iamb): based on the Markov blanket detection algorithm of the same name (Tsamardinos et al., 2003), which is based on a two-phase selection scheme (a forward selection followed by an attempt to remove false positives).
- *Fast Incremental Association* (fast.iamb): a variant of IAMB which uses speculative stepwise forward selection to reduce the number of conditional independence tests (Yaramakala and Margaritis, 2005).
- *Interleaved Incremental Association* (inter.iamb): another variant of IAMB which uses forward stepwise selection (Tsamardinos et al., 2003) to avoid false positives in the Markov blanket detection phase.

This package includes three implementations of each algorithm:

- an optimized implementation (used when the `optimized` parameter is set to `TRUE`), which uses backtracking to roughly halve the number of independence tests.
- an unoptimized implementation (used when the `optimized` parameter is set to `FALSE`) which is better at uncovering possible erratic behaviour of the statistical tests.
- a cluster-aware implementation, which requires a running cluster set up with the `makeCluster` function from the **snow** package. See snow integration for a sample usage.

The computational complexity of these algorithms is polynomial in the number of tests, usually $O(N^2)$ ($O(N^4)$ in the worst case scenario), where N is the number of variables. Execution time scales linearly with the size of the data set.

Available score-based learning algorithms

- *Hill-Climbing* (hc): a *hill climbing* greedy search on the space of the directed graphs. The optimized implementation uses score caching, score decomposability and score equivalence to reduce the number of duplicated tests.
- *Tabu Search* (tabu): a modified hill climbing able to escape local optima by selecting a network that minimally decreases the score function.

Random restart with a configurable number of perturbing operations is implemented for both algorithms.

Available hybrid learning algorithms

- *Max-Min Hill-Climbing* (mmhc): a hybrid algorithm which combines the Max-Min Parents and Children algorithm (to restrict the search space) and the Hill-Climbing algorithm (to find the optimal network structure in the restricted space).
- *Restricted Maximization* (rsmax2): a more general implementation of the Max-Min Hill-Climbing, which can use any combination of constraint-based and score-based algorithms.

Other (constraint-based) local discovery algorithms

These algorithms learn the structure of the undirected graph underlying the Bayesian network, which is known as the *skeleton* of the network or the *(partial) correlation graph*. Therefore all the arcs are undirected, and no attempt is made to detect their orientation. They are often used in hybrid learning algorithms.

- *Max-Min Parents and Children* (mmpc): a forward selection technique for neighbourhood detection based on the maximization of the minimum association measure observed with any subset of the nodes selected in the previous iterations (Tsamardinos, Brown and Aliferis, 2006).
- *Hiton Parents and Children* (si.hiton.pc): a fast forward selection technique for neighbourhood detection designed to exclude nodes early based on the marginal association. The implementation follows the Semi-Interleaved variant of the algorithm described in Aliferis et al. (2010).
- *Chow-Liu* (chow.liu): an application of the minimum-weight spanning tree and the information inequality. It learn the tree structure closest to the true one in the probability space (Chow and Liu, 1968).
- *ARACNE* (aracne): an improved version of the Chow-Liu algorithm that is able to learn polytrees (Margolin et al., 2006).

All these algorithms have three implementations (unoptimized, optimized and cluster-aware) like other constraint-based algorithms.

Bayesian Network classifiers

The algorithms are aimed at classification, and favour predictive power over the ability to recover the correct network structure. The implementation in **bnlearn** assumes that all variables, including the classifiers, are discrete.

- *Naive Bayes* (naive.bayes): a very simple algorithm assuming that all classifiers are independent and using the posterior probability of the target variable for classification.
- *Tree-Augmented Naive Bayes* (tree.bayes): a improvement over naive Bayes, this algorithms uses Chow-Liu to approximate the dependence structure of the classifiers.

Available (conditional) independence tests

The conditional independence tests used in *constraint-based* algorithms in practice are statistical tests on the data set. Available tests (and the respective labels) are:

- *discrete case* (multinomial distribution)
  - *mutual information*: an information-theoretic distance measure. It's proportional to the log-likelihood ratio (they differ by a $2n$ factor) and is related to the deviance of the tested models. The asymptotic χ^2 test (`mi`), the Monte Carlo permutation test (`mc-mi`), the sequential Monte Carlo permutation test (`smc-mi`), and the semiparametric test (`sp-mi`) are implemented.
  - *shrinkage estimator* for the *mutual information* (`mi-sh`): an improved asymptotic χ^2 test based on the James-Stein estimator for the mutual information.
  - *Pearson's X^2*: the classical Pearson's X^2 test for contingency tables. The asymptotic χ^2 test (`x2`), the Monte Carlo permutation test (`mc-x2`), the sequential Monte Carlo permutation test (`smc-x2`) and semiparametric test (`sp-x2`) are implemented.
- *continuous case* (multivariate normal distribution)
  - *linear correlation*: linear correlation. The exact Student's t test (`cor`), the Monte Carlo permutation test (`mc-cor`) and the sequential Monte Carlo permutation test (`smc-cor`) are implemented.

  - *Fisher's Z*: a transformation of the linear correlation with asymptotic normal distribution. Used by commercial software (such as TETRAD II) for the PC algorithm (an R implementation is present in the `pcalg` package on CRAN). The asymptotic normal test (`zf`), the Monte Carlo permutation test (`mc-zf`) and the sequential Monte Carlo permutation test (`smc-zf`) are implemented.
  - *mutual information*: an information-theoretic distance measure. Again it's proportional to the log-likelihood ratio (they differ by a $2n$ factor). The asymptotic χ^2 test (`mi-g`),the Monte Carlo permutation test (`mc-mi-g`) and the sequential Monte Carlo permutation test (`smc-mi-g`) are implemented.
  - *shrinkage estimator* for the *mutual information* (`mi-g-sh`): an improved asymptotic χ^2 test based on the James-Stein estimator for the mutual information.

Available network scores

Available scores (and the respective labels) are:

- *discrete case* (multinomial distribution)
  - the multinomial *log-likelihood* (`loglik`) score, which is equivalent to the *entropy measure* used in Weka.
  - the *Akaike Information Criterion* score (`aic`).
  - the *Bayesian Information Criterion* score (`bic`), which is equivalent to the *Minimum Description Length* (MDL) and is also known as *Schwarz Information Criterion*.
  - the logarithm of the *Bayesian Dirichlet equivalent* score (`bde`), a score equivalent Dirichlet posterior density.
  - the logarithm of the modified *Bayesian Dirichlet equivalent* score (`mbde`) for mixtures of experimental and observational data (not score equivalent).
  - the logarithm of the *K2* score (`k2`), a Dirichlet posterior density (not score equivalent).
- *continuous case* (multivariate normal distribution)
  - the multivariate Gaussian *log-likelihood* (`loglik-g`) score.
  - the corresponding *Akaike Information Criterion* score (`aic-g`).
  - the corresponding *Bayesian Information Criterion* score (`bic-g`).
  - a score equivalent *Gaussian posterior density* (`bge`).

Whitelist and blacklist support

All learning algorithms support arc whitelisting and blacklisting:

- blacklisted arcs are never present in the graph.
- arcs whitelisted in one direction only (i.e. $A \to B$ is whitelisted but $B \to A$ is not) have the respective reverse arcs blacklisted, and are always present in the graph.
- arcs whitelisted in both directions (i.e. both $A \to B$ and $B \to A$ are whitelisted) are present in the graph, but their direction is set by the learning algorithm.

Any arc whitelisted and blacklisted at the same time is assumed to be whitelisted, and is thus removed from the blacklist.

In algorithms that learn undirected graphs, such as ARACNE and Chow-Liu, an arc must be blacklisted in both directions to blacklist the underlying undirected arc.

Error detection and correction: the strict mode

Optimized implementations of constraint-based algorithms rely heavily on backtracking to reduce the number of tests needed by the learning algorithm. This approach may sometimes hide errors either in the Markov blanket or the neighbourhood detection steps, such as when hidden variables are present or there are external (logical) constraints on the interactions between the variables.

On the other hand, in the unoptimized implementations of constraint-based algorithms the learning of the Markov blanket and neighbourhood of each node is completely independent from the rest of the learning process. Thus it may happen that the Markov blanket or the neighbourhoods are not symmetric (i.e. A is in the Markov blanket of B but not vice versa), or that some arc directions conflict with each other.

The `strict` parameter enables some measure of error correction for such inconsistencies, which may help to retrieve a good model when the learning process would otherwise fail:

- if `strict` is set to `TRUE`, every error stops the learning process and results in an error message.
- if `strict` is set to `FALSE`:
  1. v-structures are applied to the network structure in lowest-p.value order; if any arc is already oriented in the opposite direction, the v-structure is discarded.
  2. nodes which cause asymmetries in any Markov blanket are removed from that Markov blanket; they are treated as false positives.
  3. nodes which cause asymmetries in any neighbourhood are removed from that neighbourhood; again they are treated as false positives (see Tsamardinos, Brown and Aliferis, 2006).

  Each correction results in a warning.

Author(s)

Marco Scutari
UCL Genetics Institute (UGI)
University College London

Maintainer: Marco Scutari <marco.scutari@gmail.com>

References

(a BibTeX file with all the references cited throughout this manual is present in the '`bibtex`' directory of this package)

Scutari M (2010). "Learning Bayesian Networks with the bnlearn R Package". Journal of Statistical Software, **35**(3), 1-22. URL http://www.jstatsoft.org/v35/i03/.

Aliferis FC, Statnikov A, Tsamardinos I, Subramani M, Koutsoukos XD (2010). "Local Causal and Markov Blanket Induction for Causal Discovery and Feature Selection for Classification Part I: Algorithms and Empirical Evaluation". *Journal of Machine Learning Research*, **11**, 171-234.

Chow CK, Liu CN (1968). "Approximating Discrete Probability Distributions With Dependence Trees". IEEE Transactions on InformationTheory, IT-14:462-467.

Friedman N, Geiger D, Goldszmidt M (1997). "Bayesian Network Classifiers". *Machine Learning*, **29**(2–3), 131–163.

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.

Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-03-153.

Margolin AA, Nemenman I, Basso K, Wiggins C, Stolovitzky G, Dalla Favera R, Califano A (2006). "ARACNE: An Algorithm for the Reconstruction of Gene Regulatory Networks in a Mammalian Cellular Context". *BMC Bioinformatics*, **7**(Suppl 1):S7.

Pearl J (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann.

Tsamardinos I, Aliferis CF, Statnikov A (2003). "Algorithms for Large Scale Markov Blanket Discovery". In "Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference", pp. 376-381. AAAI Press.

Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm". Machine Learning, 65(1), 31-78.

Yaramakala S, Margaritis D (2005). "Speculative Markov Blanket Discovery for Optimal Feature Selection". In "ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining", pp. 809-812. IEEE Computer Society.

Examples

```
library(bnlearn)
data(learning.test)

## Simple learning
# first try the Grow-Shrink algorithm
res = gs(learning.test)
# plot the network structure.
plot(res)
# now try the Incremental Association algorithm.
res2 = iamb(learning.test)
# plot the new network structure.
plot(res2)
# the network structures seem to be identical, don't they?
all.equal(res, res2)
# [1] TRUE
# how many tests each of the two algorithms used?
res$learning$ntests
# [1] 43
res2$learning$ntests
# [1] 50
# and the unoptimized implementation of these algorithms?
## Not run: gs(learning.test, optimized = FALSE)$learning$ntests
# [1] 93
## Not run: iamb(learning.test, optimized = FALSE)$learning$ntests
# [1] 116

## Greedy search
```

```
res = hc(learning.test)
plot(res)

## Another simple example (Gaussian data)
data(gaussian.test)
# first try the Grow-Shrink algorithm
res = gs(gaussian.test)
plot(res)

## Blacklist and whitelist use
# the arc B - F should not be there?
blacklist = data.frame(from = c("B", "F"), to = c("F", "B"))
blacklist
#   from to
# 1    B  F
# 2    F  B
res3 = gs(learning.test, blacklist = blacklist)
plot(res3)
# force E - F direction (E -> F).
whitelist = data.frame(from = c("E"), to = c("F"))
whitelist
#   from to
# 1    E  F
res4 = gs(learning.test, whitelist = whitelist)
plot(res4)
# use both blacklist and whitelist.
res5 = gs(learning.test, whitelist = whitelist, blacklist = blacklist)
plot(res5)

## Debugging
# use the debugging mode to see the learning algorithms
# in action.
res = gs(learning.test, debug = TRUE)
res = hc(learning.test, debug = TRUE)
# log the learning process for future reference.
## Not run:
sink(file = "learning-log.txt")
res = gs(learning.test, debug = TRUE)
sink()

## End(Not run)
# if something seems wrong, try the unoptimized version
# in strict mode (inconsistencies trigger errors):
## Not run:
res = gs(learning.test, optimized = FALSE, strict = TRUE, debug = TRUE)

## End(Not run)
# or disable strict mode to let the algorithm fix errors on the fly:
## Not run:
res = gs(learning.test, optimized = FALSE, strict = FALSE, debug = TRUE)

## End(Not run)
```

alarm *ALARM Monitoring System (synthetic) data set*

Description

The ALARM ("A Logical Alarm Reduction Mechanism") is a Bayesian network designed to provide an alarm message system for patient monitoring.

Usage

```
data(alarm)
```

Format

The `alarm` data set contains the following 37 variables:

- `CVP` (*central venous pressure*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `PCWP` (*pulmonary capillary wedge pressure*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `HIST` (*history*): a two-level factor with levels `TRUE` and `FALSE`.
- `TPR` (*total peripheral resistance*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `BP` (*blood pressure*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `CO` (*cardiac output*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `HRBP` (*heart rate / blood pressure*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `HREK` (*heart rate measured by an EKG monitor*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `HRSA` (*heart rate / oxygen saturation*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `PAP` (*pulmonary artery pressure*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `SAO2` (*arterial oxygen saturation*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `FIO2` (*fraction of inspired oxygen*): a two-level factor with levels `LOW` and `NORMAL`.
- `PRSS` (*breathing pressure*): a four-level factor with levels `ZERO`, `LOW`, `NORMAL` and `HIGH`.
- `ECO2` (*expelled CO2*): a four-level factor with levels `ZERO`, `LOW`, `NORMAL` and `HIGH`.
- `MINV` (*minimum volume*): a four-level factor with levels `ZERO`, `LOW`, `NORMAL` and `HIGH`.
- `MVS` (*minimum volume set*): a three-level factor with levels `LOW`, `NORMAL` and `HIGH`.
- `HYP` (*hypovolemia*): a two-level factor with levels `TRUE` and `FALSE`.
- `LVF` (*left ventricular failure*): a two-level factor with levels `TRUE` and `FALSE`.
- `APL` (*anaphylaxis*): a two-level factor with levels `TRUE` and `FALSE`.
- `ANES` (*insufficient anesthesia/analgesia*): a two-level factor with levels `TRUE` and `FALSE`.
- `PMB` (*pulmonary embolus*): a two-level factor with levels `TRUE` and `FALSE`.
- `INT` (*intubation*): a three-level factor with levels `NORMAL`, `ESOPHAGEAL` and `ONESIDED`.

- KINK (*kinked tube*): a two-level factor with levels TRUE and FALSE.
- DISC (*disconnection*): a two-level factor with levels TRUE and FALSE.
- LVV (*left ventricular end-diastolic volume*): a three-level factor with levels LOW, NORMAL and HIGH.
- STKV (*stroke volume*): a three-level factor with levels LOW, NORMAL and HIGH.
- CCHL (*catecholamine*): a two-level factor with levels NORMAL and HIGH.
- ERLO (*error low output*): a two-level factor with levels TRUE and FALSE.
- HR (*heart rate*): a three-level factor with levels LOW, NORMAL and HIGH.
- ERCA (*electrocauter*): a two-level factor with levels TRUE and FALSE.
- SHNT (*shunt*): a two-level factor with levels NORMAL and HIGH.
- PVS (*pulmonary venous oxygen saturation*): a three-level factor with levels LOW, NORMAL and HIGH.
- ACO2 (*arterial CO2*): a three-level factor with levels LOW, NORMAL and HIGH.
- VALV (*pulmonary alveoli ventilation*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VLNG (*lung ventilation*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VTUB (*ventilation tube*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VMCH (*ventilation machine*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.

Note

The R script to generate data from this network is shipped in the 'network.scripts' directory of this package.

Source

Beinlich I, Suermondt HJ, Chavez RM, Cooper GF (1989). "The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks." In "Proceedings of the 2nd European Conference on Artificial Intelligence in Medicine", pp. 247-256. Springer-Verlag.

Elidan G (2001). "Bayesian Network Repository".
http://www.cs.huji.ac.il/site/labs/compbio/Repository.

Examples

```
# load the data and build the correct network from the model string.
data(alarm)
res = empty.graph(names(alarm))
modelstring(res) = paste("[HIST|LVF][CVP|LVV][PCWP|LVV][HYP][LVV|HYP:LVF]",
  "[LVF][STKV|HYP:LVF][ERLO][HRBP|ERLO:HR][HREK|ERCA:HR][ERCA]",
  "[HRSA|ERCA:HR][ANES][APL][TPR|APL][ECO2|ACO2:VLNG][KINK]",
  "[MINV|INT:VLNG][FIO2][PVS|FIO2:VALV][SAO2|PVS:SHNT][PAP|PMB][PMB]",
  "[SHNT|INT:PMB][INT][PRSS|INT:KINK:VTUB][DISC][MVS][VMCH|MVS]",
  "[VTUB|DISC:VMCH][VLNG|INT:KINK:VTUB][VALV|INT:VLNG][ACO2|VALV]",
  "[CCHL|ACO2:ANES:SAO2:TPR][HR|CCHL][CO|HR:STKV][BP|CO:TPR]", sep = "")
## Not run:
```

```
# there are too many nodes for plot(), use graphviz.plot().
graphviz.plot(res)
## End(Not run)
```

`arc operations` *Drop, add or set the direction of an arc*

Description

Drop, add or set the direction of an arc.

Usage

```
set.arc(x, from, to, check.cycles = TRUE, debug = FALSE)
drop.arc(x, from, to, debug = FALSE)
reverse.arc(x, from, to, check.cycles = TRUE, debug = FALSE)
```

Arguments

`x`	an object of class `bn`.
`from`	a character string, the label of a node.
`to`	a character string, the label of another node.
`check.cycles`	a boolean value. If `TRUE` the graph is tested for acyclicity; otherwise the graph is returned anyway.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The `set.arc` function operates in the following way:

- if there is no arc between `from` and `to`, the arc `from` $\rightarrow$ `to` is added.
- if there is an undirected arc between `from` and `to`, its direction is set to `from` $\rightarrow$ `to`.
- if the arc `to` $\rightarrow$ `from` is present, it's reversed.
- if the arc `from` $\rightarrow$ `to` is present, no action is taken.

The `drop.arc` function operates in the following way:

- if there is no arc between `from` and `to`, no action is taken.
- if there is an undirected arc between `from` and `to`, it's dropped.
- if there is a directed arc between `from` and `to`, it's dropped regardless of its direction.

The `reverse.arc` function operates in the following way:

- if there is no arc between `from` and `to`, it returns an error.
- if there is an undirected arc between `from` and `to`, it returns an error.
- if the arc `to` $\rightarrow$ `from` is present, it's reversed.
- if the arc `from` $\rightarrow$ `to` is present, it's reversed.

Value

set.arc and drop.arc return invisibly an updated copy of x.

Author(s)

Marco Scutari

Examples

```
data(learning.test)
res = gs(learning.test)

## use debug = TRUE for more information
## Not run:
set.arc(res, "A", "B", debug = TRUE)
drop.arc(res, "A", "B", debug = TRUE)
reverse.arc(res, "A", "D", debug = TRUE)

## End(Not run)
```

---

arc.strength *Measure arc strength*

Description

Measure the strength of the probabilistic relationships expressed by the arcs of a Bayesian network, and use model averaging to build a network containing only the significant arcs.

Usage

```
# strength of the arcs present in x.
arc.strength(x, data, criterion = NULL, ..., debug = FALSE)
# strength of all possible arcs, as learned from bootstrapped data.
boot.strength(data, R = 200, m = nrow(data),
  algorithm, algorithm.args = list(), cpdag = TRUE, debug = FALSE)
# strength of all possible arcs, from a list of custom networks.
custom.strength(networks, nodes, weights = NULL, cpdag = TRUE,
  debug = FALSE)

# averaged network structure.
averaged.network(strength, nodes, threshold)
```

Arguments

x	an object of class bn.
networks	a list, containing either object of class bn or arc sets (matrices or data frames with two columns, optionally labeled "from" and "to").

data	a data frame containing the data the Bayesian network was learned from.
strength	an object of class bn.strength, see below.
threshold	a numeric value, the minimum strength required for an arc to be included in the averaged network. The default value is the threshold attribute of the strength argument.
nodes	a vector of character strings, the labels of the nodes in the network. In averaged.network, it defaults to the set of the unique node labels in the strength argument.
criterion	a character string, the label of a score function, the label of an independence test or bootstrap. See bnlearn-package for details on the first two possibilities.
R	a positive integer, the number of bootstrap replicates.
m	a positive integer, the size of each bootstrap replicate.
weights	a vector of non-negative numbers, to be used as weights when averaging network structures to compute strength coefficients. If NULL, weights are assumed to be uniform.
cpdag	a boolean value. If TRUE the (PDAG of) the equivalence class is used instead of the network structure itself. It should make it easier to identify score-equivalent arcs.
algorithm	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are gs, iamb, fast.iamb, inter.iamb, mmpc, hc, tabu, mmhc and rsmax2. See bnlearn-package and the documentation of each algorithm for details.
algorithm.args	a list of extra arguments to be passed to the learning algorithm.
...	additional tuning parameters for the network score (if criterion is the label of a score function, see score for details), the conditional independence test (currently the only one is B, the number of permutations) or the bootstrap simulation (if criterion is set to bootstrap, see boot.strength for details).
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Details

If criterion is a conditional independence test, the strength is a p-value (so the lower the value, the stronger the relationship). The only possible additional parameter is B, the number of permutations to be generated for each permutation test.

If criterion is the label of a score function, the strength is measured by the score gain/loss which would be caused by the arc's removal. There may be additional parameters depending on the choice of the score, see score for details.

If criterion is bootstrap, the strength is computed as in boot.strength. The additional parameters are R, m, algorithm and algorithm.args; if the latter two are not specified, the values stored in x are used.

Model averaging is supported for objects of class bn.strength returned by boot.strength, by custom.strength, or by arc.strength with criterion set to bootstrap. The returned network contains the arcs whose strength is greater than the threshold attribute of the bn.strength object passed to averaged.network.

Value

arc.strength, boot.strength and custom.strength return an object of class bn.strength; boot.strength and custom.strength also include information about the relative probabilities of arc directions.

averaged.network returns an object of class bn.

See bn.strength class and bn-class for details.

Author(s)

Marco Scutari

References

for model averaging and boostrap strength (confidence):

Friedman N, Goldszmidt M, Wyner A (1999). "Data Analysis with Bayesian Networks: A Bootstrap Approach". In "UAI '99: Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence", pp. 196-20. Morgan Kaufmann.

for the computation of the strength (confidence) significance threshold:

Scutari M, Nagarajan R (2011). "On Identifying Significant Edges in Graphical Models". In "Proceedings of the Workshop 'Probabilistic Problem Solving in Biomedicine' of the 13th Artificial Intelligence in Medicine (AIME) Conference", pp. 15-27.

See Also

strength.plot, choose.direction, score, ci.test.

Examples

```
data(learning.test)
res = gs(learning.test)
res = set.arc(res, "A", "B")
arc.strength(res, learning.test)
#   from to      strength
# 1    A  B  0.000000e+00
# 2    A  D  0.000000e+00
# 3    B  E 1.024198e-320
# 4    C  D  0.000000e+00
# 5    F  E 3.935648e-245
arcs = boot.strength(learning.test, algorithm = "hc")
arcs[(arcs$strength > 0.85) & (arcs$direction >= 0.5), ]
#    from to strength direction
# 1     A  B        1       0.5
# 3     A  D        1       1.0
# 6     B  A        1       0.5
# 9     B  E        1       1.0
# 13    C  D        1       1.0
# 30    F  E        1       1.0
averaged.network(arcs)
#
```

```
#   Random/Generated Bayesian network
#
#   model:
#    [A][C][F][B|A][D|A:C][E|B:F]
#   nodes:                                 6
#   arcs:                                  5
#     undirected arcs:                     0
#     directed arcs:                       5
#   average markov blanket size:           2.33
#   average neighbourhood size:            1.67
#   average branching factor:              0.83
#
#   generation algorithm:                  Model Averaging
#   significance threshold:                0.025

## Not run:
start = random.graph(nodes = names(learning.test), num = 50)
netlist = lapply(start, function(net) {
  hc(learning.test, score = "bde", iss = 10, start = net) })
arcs = custom.strength(netlist, nodes = names(learning.test),
         cpdag = FALSE)
arcs[(arcs$strength > 0.85) & (arcs$direction >= 0.5), ]
#    from to strength direction
# 1     A  B        1      1.00
# 3     A  D        1      1.00
# 9     B  E        1      0.98
# 13    C  D        1      0.96
# 30    F  E        1      0.66
modelstring(averaged.network(arcs))
# [1] "[A][C][F][B|A][D|A:C][E|B:F]"

## End(Not run)
```

---

asia *Asia (synthetic) data set by Lauritzen and Spiegelhalter*

Description

Small synthetic data set from Lauritzen and Spiegelhalter (1988) about lung diseases (tuberculosis, lung cancer or bronchitis) and visits to Asia.

Usage

```
data(asia)
```

Format

The `asia` data set contains the following variables:

- `D` (*dyspnoea*), a two-level factor with levels `yes` and `no`.

- T (*tuberculosis*), a two-level factor with levels yes and no.
- L (*lung cancer*), a two-level factor with levels yes and no.
- B (*bronchitis*), a two-level factor with levels yes and no.
- A (*visit to Asia*), a two-level factor with levels yes and no.
- S (*smoking*), a two-level factor with levels yes and no.
- X (*chest X-ray*), a two-level factor with levels yes and no.
- E (*tuberculosis versus lung cancer/bronchitis*), a two-level factor with levels yes and no.

Note

Lauritzen and Spiegelhalter (1988) motivate this example as follows:

“Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer or bronchitis, or none of them, or more than one of them. A recent visit to Asia increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and bronchitis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis, as neither does the presence or absence of dyspnoea.”

Standard learning algorithms are not able to recover the true structure of the network because of the presence of a node (E) with conditional probabilities equal to both 0 and 1. Monte Carlo tests seems to behvae better than their parametric counterparts.

The R script to generate data from this network is shipped in the ‘network.scripts’ directory of this package.

Source

Lauritzen S, Spiegelhalter D (1988). "Local Computation with Probabilities on Graphical Structures and their Application to Expert Systems (with discussion)". *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **50**(2), 157-224.

Examples

```
# load the data and build the correct network from the model string.
data(asia)
res = empty.graph(names(asia))
modelstring(res) = "[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]"
plot(res)
```

---

bn class *The bn class structure*

Description

The structure of an object of the bn S3 class.

Details

An object of class bn is a list containing at least the following components:

- learning: a list containing some information about the results of the learning algorithm. It's never changed afterward.
  - whitelist: a sanitized copy of the whitelist parameter (a two-column matrix, whose columns are labeled from and to).
  - blacklist: a sanitized copy of the blacklist parameter (a two-column matrix, whose columns are labeled from and to).
  - test: the label of the conditional independence test used by the learning algorithm (a character string). The label of the network score is used for score-based and hybrid algorithms, and "none" for randomly generated graphs.
  - ntests: the number of conditional independence tests or score comparisons used in the learning (an integer value).
  - algo: the label of the learning algorithm or the random generation algorithm used to generate the network (a character string).
  - args: a list. The values of the parameters of either the conditional tests or the scores used in the learning process. Only the relevant ones are stored, so this may be an empty list.
    - alpha: the target nominal type I error rate (a numeric value) of the conditional independence tests.
    - iss: a positive numeric value, the imaginary sample size used by the bge and bde scores.
    - phi: a character string, either heckerman or bottcher; used by the bge score.
    - k: a positive numeric value, the penalty per parameter used by the aic, aic-g, bic and bic-g scores.
    - prob: the probability of each arc to be present in a graph generated by the ordered graph generation algorithm.
    - burn.in: the number of iterations for the ic-dag graph generation algorithm to converge to a stationary (and uniform) probability distribution.
    - max.degree: the maximum degree for any node in a graph generated by the ic-dag graph generation algorithm.
    - max.in.degree: the maximum in-degree for any node in a graph generated by the ic-dag graph generation algorithm.
    - max.out.degree: the maximum out-degree for any node in a graph generated by the ic-dag graph generation algorithm.
    - training: a character string, the label of the training node in a Bayesian network classifier.
- nodes: a list. Each element is named after a node and contains the following elements:
  - mb: the Markov blanket of the node (a vector of character strings).
  - nbr: the neighbourhood of the node (a vector of character strings).
  - parents: the parents of the node (a vector of character strings).
  - children: the children of the node (a vector of character strings).
- arcs: the arcs of the Bayesian network (a two-column matrix, whose columns are labeled from and to). Undirected arcs are stored as two directed arcs with opposite directions between the corresponding incident nodes.

Additional (optional) components under `learning`:

- `optimized`: whether additional optimizations have been used in the learning algorithm (a boolean value).
- `restrict`: the label of the constraint-based algorithm used in the “Restrict” phase of a hybrid learning algorithm (a character string).
- `rtest`: the label of the conditional independence test used in the “Restrict” phase of a hybrid learning algorithm (a character string).
- `maximize`: the label of the score-based algorithm used in the “Maximize” phase of a hybrid learning algorithm (a character string).
- `maxscore`: the label of the network score used in the “Maximize” phase of a hybrid learning algorithm (a character string).

Author(s)

Marco Scutari

`bn.boot` *Parametric and nonparametric bootstrap of Bayesian networks*

Description

Apply a user-specified function to the Bayesian network structures learned from bootstrap samples of the original data.

Usage

```
bn.boot(data, statistic, R = 200, m = nrow(data),
  sim = "ordinary", algorithm, algorithm.args = list(),
  statistic.args = list(), cluster = NULL, debug = FALSE)
```

Arguments

`data`	a data frame containing the variables in the model.
`statistic`	a function or a character string (the name of a function) to be applied to each bootstrap replicate.
`R`	a positive integer, the number of bootstrap replicates.
`m`	a positive integer, the size of each bootstrap replicate.
`sim`	a character string indicating the type of simulation required. Possible values are `"ordinary"` (the default) and `"parametric"`.
`algorithm`	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are `gs`, `iamb`, `fast.iamb`, `inter.iamb`, `mmpc`, `hc`, `tabu`, `mmhc` and `rsmax2`. See `bnlearn-package` and documentation of each algorithm for details.

`algorithm.args` a list of extra arguments to be passed to the learning algorithm.

`statistic.args` a list of extra arguments to be passed to the function specified by `statistic`.

`cluster` an optional cluster object from package **snow**. See `snow integration` for details and a simple example.

`debug` a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The first argument of `statistic` is the `bn` object encoding the network structure learned from the bootstrap sample; the arguments specified in `statistics.args` are extracted from the list and passed to `statitstics` as the 2nd, 3rd, etc. arguments.

Value

A list containing the results of the calls to `statistic`.

Author(s)

Marco Scutari

References

Friedman N, Goldszmidt M, Wyner A (1999). "Data Analysis with Bayesian Networks: A Bootstrap Approach". In "UAI '99: Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence", pp. 196-20. Morgan Kaufmann.

See Also

`bn.cv`, `rbn`.

Examples

```
## Not run:
data(learning.test)
bn.boot(data = learning.test, R = 2, m = 500, algorithm = "gs",
  statistic = arcs)
# [[1]]
#      from to
# <arcs for the first replicate>
#
# [[2]]
#      from to
# <arcs for the second replicate>

## End(Not run)
```

bn.cv *Cross-validation for Bayesian networks*

Description

Perform a k-fold cross-validation for a learning algorithm or a fixed network structure.

Usage

```
bn.cv(data, bn, loss = NULL, k = 10, algorithm.args = list(),
  loss.args = list(), fit = "mle", fit.args = list(),
  cluster = NULL, debug = FALSE)
```

Arguments

`data`	a data frame containing the variables in the model.
`bn`	either a character string (the label of the learning algorithm to be applied to the training data in each iteration) or an object of class `bn` (a fixed network structure).
`loss`	a character string, the label of a loss function. If none is specified, the default loss function is the *Log-Likelihood Loss* for both discrete and continuous data sets. See below for additional details.
`k`	a positive integer number, the number of groups into which the data will be split.
`algorithm.args`	a list of extra arguments to be passed to the learning algorithm.
`loss.args`	a list of extra arguments to be passed to the loss function specified by `loss`.
`fit`	a character string, the label of the method used to fit the parameters of the newtork. See `bn.fit` for details.
`fit.args`	additional arguments for the parameter estimation prcoedure, see again `bn.fit` for details..
`cluster`	an optional cluster object from package **snow**. See `snow integration` for details and a simple example.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The following loss functions are implemented:

- *Log-Likelihood Loss* (`logl`): also known as *negative entropy* or *negentropy*, it's the negated expected log-likelihood of the test set for the Bayesian network fitted from the training set.
- *Gaussian Log-Likelihood Loss* (`logl-g`): the negated expected log-likelihood for Gaussian Bayesian networks.
- *Classification Error* (`pred`): the *prediction error* for a single node (specified by the `target` parameter in `loss.args`) in a discrete network.

Value

An object of class `bn.kcv`.

Author(s)

Marco Scutari

References

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

See Also

`bn.boot`, `rbn`, `bn.kcv-class`.

Examples

```
bn.cv(learning.test, 'hc', loss = "pred", loss.args = list(target = "F"))
#
#  k-fold cross-validation for Bayesian networks
#
#  target learning algorithm:             Hill-Climbing
#  number of subsets:                     10
#  loss function:                         Classification Error
#  expected loss:                         0.509
#
bn.cv(gaussian.test, 'mmhc')
#
#  k-fold cross-validation for Bayesian networks
#
#  target learning algorithm:             Max-Min Hill Climbing
#  number of subsets:                     10
#  loss function:                         Log-Likelihood Loss (Gaussian)
#  expected loss:                         10.63062
#
```

`bn.fit` *Fit the parameters of a Bayesian network*

Description

Fit the parameters of a Bayesian network conditional on its structure.

Usage

```
bn.fit(x, data, method = "mle", ..., debug = FALSE)
custom.fit(x, dist)
bn.net(x, debug = FALSE)
```

Arguments

x	an object of class bn (for bn.fit and custom.fit) or an object of class bn.fit (for bn.net).
data	a data frame containing the variables in the model.
dist	a named list, with element for each node of x. See below.
method	a character string, either mle for *Maximum Likelihood parameter estimation* or bayes for *Bayesian parameter estimation* (currently implemented only for discrete data).
...	additional arguments for the parameter estimation prcoedure, see below.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Details

bn.fit fits the parameters of a Bayesian network given its structure and a data set; bn.net returns the network structure underlying a fitted network.

An in-place replacement method is available to change the parameters of each node in a bn.fit object; see the examples for both discrete and continuous networks below. For a node in a discrete network, the new parameters must be in a table object. For a node in a continuous network, the new parameters can be defined either by an lm, glm or pensim object (the latter is from the penalized package) or in a list with elements named coef, sd and optionally fitted and resid.

custom.fit takes a set of user-specified distributions and their parameters and uses them to build a bn.fit object. Its purpose is to specify a Bayesian network (complete with the parameters, not only the structure) using knowledge from experts in the field instead of learning it from a data set. The distributions must be passed to the function in a list, with elements named after the nodes of the network structure x. Each element of the list must be in one of the formats described above for in-place replacement.

Value

bn.fit returns an object of class bn.fit, bn.net an object of class bn. See bn class and bn.fit class for details.

Note

Due to the way Bayesian networks are defined it's possible to estimate their parameters only if the network structure is completely directed (i.e. there are no undirected arcs). See set.arc and pdag2dag for two ways of manually setting the direction of one or more arcs.

The only supported additional parameter is the imaginary sample size (iss) for the Dirichlet posterior distribution of discrete networks (see score for details).

Author(s)

Marco Scutari

See Also

bn.fit utilities, bn.fit plots.

Examples

```
data(learning.test)

# learn the network structure.
res = gs(learning.test)
# set the direction of the only undirected arc, A - B.
res = set.arc(res, "A", "B")
# estimate the parameters of the Bayesian network.
fitted = bn.fit(res, learning.test)
# replace the parameters of the node B.
new.cpt = matrix(c(0.1, 0.2, 0.3, 0.2, 0.5, 0.6, 0.7, 0.3, 0.1),
            byrow = TRUE, ncol = 3,
            dimnames = list(B = c("a", "b", "c"), A = c("a", "b", "c")))
fitted$B = as.table(new.cpt)

# learn the network structure.
res = hc(gaussian.test)
# estimate the parameters of the Bayesian network.
fitted = bn.fit(res, gaussian.test)
# replace the parameters of the node F.
fitted$F = list(coef = c(1, 2, 3, 4, 5), sd = 3)
# set again the original parameters
fitted$F = lm(F ~ A + D + E + G, data = gaussian.test)

# discrete Bayesian network from expert knowledge.
net = model2network("[A][B][C|A:B]")
cptA = matrix(c(0.4, 0.6), ncol = 2, dimnames = list(NULL, c("LOW", "HIGH")))
cptB = matrix(c(0.8, 0.2), ncol = 2, dimnames = list(NULL, c("GOOD", "BAD")))
cptC = c(0.5, 0.5, 0.4, 0.6, 0.3, 0.7, 0.2, 0.8)
dim(cptC) = c(2, 2, 2)
dimnames(cptC) = list("C" = c("TRUE", "FALSE"), "A" =  c("LOW", "HIGH"), "B" = c("GOOD", "BAD"))
custom.fit(net, dist = list(A = cptA, B = cptB, C = cptC))

# Gaussian Bayesian network from expert knowledge.
distA = list(coef = c("(Intercept)" = 2), sd = 1)
distB = list(coef = c("(Intercept)" = 1), sd = 1.5)
distC = list(coef = c("(Intercept)" = 0.5, "A" = 0.75, "B" = 1.32), sd = 04)
custom.fit(net, dist = list(A = distA, B = distB, C = distC))
```

---

bn.fit class *The bn.fit class structure*

Description

The structure of an object of the bn.fit S3 class.

Details

An object of class bn.fit is a list whose elements correspond to the nodes of the Bayesian network. If the latter is discrete (i.e. the nodes are multinomial random variables) each node has class bn.fit.dnode and contains the following elements:

- node: the label of the node.
- parents: the labels of the parents of the node.
- children: the labels of the children of the node.
- prob: the conditional probability table of the node given its parents.

If on the other hand the network is continuous (i.e. the nodes are Gaussian random variables) each node has class bn.fit.gnode and contains the following elements:

- node: the label of the node.
- parents: the labels of the parents of the node.
- children: the labels of the children of the node.
- coefficients: the linear regression coefficients of the parents against the node.
- residuals: the residuals of the linear regression, that is response minus fitted values.
- fitted.values: the fitted mean values of the linear regression.
- sd: the standard deviation of the residuals.

Furthermore, Bayesian network classifiers store the label of the training node in an additional attribute named training.

Author(s)

Marco Scutari

---

bn.fit plots — *Plot fitted Bayesian networks*

Description

Plot functions for the bn.fit, bn.fit.dnode and bn.fit.gnode classes, based on the **lattice** package.

Usage

```
## for Gaussian Bayesian networks.
bn.fit.qqplot(fitted, xlab = "Theoretical Quantiles",
  ylab = "Sample Quantiles", main = "Normal Q-Q Plot", ...)
bn.fit.histogram(fitted, density = TRUE, xlab = "Residuals",
  ylab = ifelse(density, "Density", ""),
  main = "Histogram of the residuals", ...)
bn.fit.xyplot(fitted, xlab = "Fitted values",
```

```
  ylab = "Residuals", main = "Residuals vs Fitted", ...)
## for discrete Bayesian networks
bn.fit.barchart(fitted, xlab = "Probabilities",
  ylab = "Levels", main = "Conditional Probabilities", ...)
bn.fit.dotplot(fitted, xlab = "Probabilities",
  ylab = "Levels", main = "Conditional Probabilities", ...)
```

Arguments

`fitted`	an object of class `bn.fit`, `bn.fit.dnode` or `bn.fit.gnode`.
`xlab, ylab, main`	the label of the x axis, of the y axis, and the plot title.
`density`	a boolean value. If `TRUE` the histogram is plotted using relative frequencies, and the matching normal density is added to the plot.
`...`	additional arguments to be passed to **lattice** functions.

Details

`bn.fit.qqplot` draws a quantile-quantile plot of the residuals.

`bn.fit.histogram` draws a histogram of the residuals, using either absolute or relative frequencies.

`bn.fit.xyplot` plots the residuals versus the fitted values.

`bn.fit.barchart` and `bn.fit.dotplot` plot the probabilities in the conditional probability table associated with each node.

Value

The **lattice** plot objects. Note that if auto-printing is turned off (for example when the code is loaded with the `source` function), the return value must be printed explicitly for the plot to be displayed.

Author(s)

Marco Scutari

See Also

`bn.fit`, `bn.fit class`.

`bn.fit utilities` *Utilities to manipulate fitted Bayesian networks*

Description

Assign, extract or compute various quantities of interest from an object of class `bn.fit`, `bn.fit.dnode` or `bn.fit.gnode`.

Usage

```
## methods available for "bn.fit"
## S3 method for class 'bn.fit'
fitted(object, ...)
## S3 method for class 'bn.fit'
coef(object, ...)
## S3 method for class 'bn.fit'
residuals(object, ...)
## S3 method for class 'bn.fit'
predict(object, node, data, ..., debug = FALSE)
## S3 method for class 'bn.fit'
logLik(object, data, ...)
## S3 method for class 'bn.fit'
AIC(object, data, ..., k = 1)
## S3 method for class 'bn.fit'
BIC(object, data, ...)

## methods available for "bn.fit.dnode"
## S3 method for class 'bn.fit.dnode'
coef(object, ...)
## S3 method for class 'bn.fit.dnode'
predict(object, data, ..., debug = FALSE)

## methods available for "bn.fit.gnode"
## S3 method for class 'bn.fit.gnode'
fitted(object, ...)
## S3 method for class 'bn.fit.gnode'
coef(object, ...)
## S3 method for class 'bn.fit.gnode'
residuals(object, ...)
## S3 method for class 'bn.fit.gnode'
predict(object, data, ..., debug = FALSE)
```

Arguments

`object`	an object of class `bn.fit`, `bn.fit.dnode` or `bn.fit.gnode`.
`node`	a character string, the label of a node.
`data`	a data frame containing the variables in the model.
`...`	additional arguments (currently ignored).
`k`	a numeric value, the penalty per parameter to be used; the default `k = 1` gives the expression used to compute AIC.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

`coef` (and its alias `coefficients`) extracts model coefficients (which are conditional probabilities in discrete networks and linear regression coefficients in Gaussian networks).

`residuals` (and its alias `resid`) extracts model residuals and `fitted` (and its alias `fitted.values`) extracts fitted values from fitted Gaussian networks.

`predict` returns the predicted values for `node` for the data specified by `data`.

Value

`predict` returns a numeric vector (for Gaussian networks) or a factor (for discrete networks).

All the other functions return a list with an element for each node in the network (if `object` has class `bn.fit`) or a numeric vector (if `object` has class `bn.fit.dnode` or `bn.fit.gnode`).

Note

Ties in prediction are broken using *Bayesian tie breaking*, i.e. sampling at random from the tied values. Therefore, setting the random seed is required to get reproducible results.

Author(s)

Marco Scutari

See Also

`bn.fit`, `bn.fit-class`.

Examples

```
data(gaussian.test)
res = hc(gaussian.test)
fitted = bn.fit(res, gaussian.test)

coefficients(fitted)
# $A
# (Intercept)
#    1.007493
#
# $B
# (Intercept)
#    2.039499
#
# $C
# (Intercept)           A           B
#    2.001083    1.995901    1.999108
#
# $D
# (Intercept)           B
#    5.995036    1.498395
#
# $E
# (Intercept)
#    3.493906
#
# $F
```

```
#  (Intercept)            A            D            E            G
# -0.006047321  1.994853041  1.005636909  1.002577002  1.494373265
#
# $G
# (Intercept)
#    5.028076
#
coefficients(fitted$C)
# (Intercept)           A           B
#    2.001083    1.995901    1.999108
str(residuals(fitted))
# List of 7
#  $ A: num [1:5000] 0.106 -1.255 0.847 -0.174 -0.519 ...
#  $ B: num [1:5000] -0.107 9.295 0.993 1.818 2.473 ...
#  $ C: num [1:5000] -1.01 0.183 -0.677 -0.153 -1.997 ...
#  $ D: num [1:5000] -0.23 0.377 0.518 0.162 -0.22 ...
#  $ E: num [1:5000] -2.612 3.546 0.341 -2.488 0.591 ...
#  $ F: num [1:5000] -0.861 1.271 -0.262 -0.479 -0.782 ...
#  $ G: num [1:5000] 4.1883 -1.3492 -2.6036 1.0574 0.0895 ...

data(learning.test)
res2 = hc(learning.test)
fitted2 = bn.fit(res2, learning.test)

coefficients(fitted2$E)
# , , F = a
#
#    B
# E        a      b      c
#   a 0.1902 0.0126 0.0244
#   b 0.0230 0.0110 0.0234
#   c 0.0230 0.0376 0.1566
#
# , , F = b
#
#    B
# E        a      b      c
#   a 0.0946 0.0166 0.0498
#   b 0.1158 0.0192 0.1062
#   c 0.0258 0.0166 0.0536
```

bn.kcv class *The bn.kcv class structure*

Description

The structure of an object of the `bn.kcv` S3 class.

Details

An object of class `bn.kcv` is a list whose elements correspond to the iterations of a k-fold cross-validation. Each element contains the following objects:

- `test`: an integer vector, the indexes of the observations used as a test set.
- `fitted`: an object of class `bn.fit`, the Bayesian network fitted from the training set.
- `loss`: the value of the loss function.

If the loss function is some form of prediction error, each element also contains:

- `predicted`: a factor, the predicted values for the test set.

In addition, an object of class `bn.kcv` has the following attributes:

- `loss`: a character string, the label of the loss function.
- `mean`: the mean of the values of the loss function computed in the `k` iterations of the cross-validation.
- `bn`: either a character string (the label of the learning algorithm to be applied to the training data in each iteration) or an object of class `bn` (a fixed network structure).

Author(s)

Marco Scutari

`bn.strength class` *The bn.strength class structure*

Description

The structure of an object of the `bn.strength` S3 class.

Details

An object of class `bn.strength` is a data frame with the following columns (one row for each arc):

- `from, to`: the nodes incident on the arc.
- `strength`: the strength of the arc. See `arc.strength`, `boot.strength`, `custom.strength` and `strength.plot` for details.

and some additional attributes:

- `mode`: a character string, the criterion used to compute the strength coefficient. It can be equal to `test`, `score` or `bootstrap`.
- `threshold`: a numeric value, the threshold used to determine if a strength coefficient is significant.

An optional column called `direction` may also be present, giving the probability of the direction of an arc given its presence in the graph.

No method is defined for this class; therefore it can be manipulated as a standard data frame.

Author(s)

Marco Scutari

`bn.var` *Structure variability of Bayesian networks*

Description

Measure the variability of the structure of a Bayesian network.

Usage

```
# first and second moments' estimation
bn.moments(data, R = 200, m = nrow(data), algorithm,
  algorithm.args = list(), reduce = NULL, debug = FALSE)
# descriptive statistics
bn.var(x, method)
```

Arguments

`data`	a data frame containing the variables in the model.
`R`	a positive integer, the number of bootstrap replicates (in `bn.moments`) or the number of Monte Carlo samples (in `bn.var.test`).
`m`	a positive integer, the bootstrap sample size.
`algorithm`	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are `gs`, `iamb`, `fast.iamb`, `inter.iamb`, `mmpc`, `hc`, `tabu`, `mmhc` and `rsmax2`. See `bnlearn-package` and the documentation of each algorithm for details.
`algorithm.args`	a list of extra arguments to be passed to the learning algorithm.
`x`	a covariance matrix or an object of class `mvber.moments` (the return value of the `bn.moments` function).
`method`	a character string, the label of the statistic. Possible values are `tvar` (*total variance*), `gvar` (`generalized variance`), `nvar` (*Frobenius matrix norm*, which is equivalent to *Nagao's test*) and `nvark` (another measure based on the *Frobenius matrix norm*).
`reduce`	a character string, either `first` or `second`. If `first` all the arcs with first moment equal to zero are dropped; if if `second` all the arcs with zero variance are dropped.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

`bn.moments` returns an object of class `mvber.moments`.

`bn.var` returns a vector of two elements, the observed value of the statistic (named `statistic`) and its normalized equivalent (named `normalized`).

Note

These functions are experimental implementations of techniques still in development; their form (name, parameters, etc.) will likely change without notice in the future.

Author(s)

Marco Scutari

References

Scutari M (2009). "Structure Variability in Bayesian Networks". *ArXiv Statistics - Methodology e-prints*. http://arxiv.org/abs/0909.1685.

Examples

```
## Not run:
z =  bn.moments(learning.test, algorithm = "gs", R = 100)
bn.var(z, method = "tvar")
#  statistic normalized
#    1.29060    0.34416

## End(Not run)
```

`choose.direction` *Try to infer the direction of an undirected arc*

Description

Check both possible directed arcs for existence, and choose the one with the lowest p-value, the highest score or the highest bootstrap probability.

Usage

```
choose.direction(x, arc, data, criterion = NULL, ...,
  debug = FALSE)
```

Arguments

`x`	an object of class bn.
`arc`	a character string vector of length 2, the labels of two nodes of the graph.
`data`	a data frame containing the data the Bayesian network was learned from.
`criterion`	a character string, the label of a score function, the label of an independence test or `bootstrap`. See `bnlearn-package` for details on the first two possibilities.
`...`	additional tuning parameters for the network score. See `score` for details.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

If `criterion` is `bootstrap`, `choose.directions` accepts the same arguments as `boot.strength`: `R` (the number of bootstrap replicates), `m` (the bootstrap sample size), `algorithm` (the structure learning algorithm), `algorithm.args` (the arguments to pass to the structure learning algorithm) and cpdag (whether to transform the network structure to the CPDAG representation of the equivalence class it belongs to).

Value

`choose.direction` returns invisibly an updated copy of `x`.

Author(s)

Marco Scutari

See Also

`score`, `arc.strength`.

Examples

```
data(learning.test)
res = gs(learning.test)

## the arc A - B has no direction.
choose.direction(res, learning.test, arc = c("A", "B"), debug = TRUE)
# * testing A - B for direction.
#   > testing A -> B with conditioning set '  '.
#     > p-value is 0 .
#   > testing B -> A with conditioning set '  '.
#     > p-value is 0 .
#   @ nothing to do, same p-value.

## let's see score equivalence in action.
choose.direction(res, learning.test, criterion = "aic",
  arc = c("A", "B"), debug = TRUE)
# * testing A - B for direction.
#   > initial score for node A is -5495.051 .
#   > initial score for node B is -4834.284 .
#   > score delta for arc A -> B is 1166.914 .
#   > score delta for arc B -> A is 1166.914 .
#   @ nothing to do, same score delta.

## arcs which introduce cycles are handled correctly.
res = set.arc(res, "A", "B")
# now A -> B -> E -> A is a cycle.
choose.direction(res, learning.test, arc = c("E", "A"), debug = TRUE)
# * testing E - A for direction.
#   > testing E -> A with conditioning set '  '.
#     > p-value is 1.426725e-99 .
#   > testing A -> E with conditioning set ' B F '.
```

```
#     > p-value is 0.9818423 .
#   > adding E -> A creates cycles!.
#   > arc A -> E isn't good, either.

## Not run:
choose.direction(res, learning.test, arc = c("D", "E"), criterion = "bootstrap",
  R = 100, algorithm = "iamb", algorithm.args = list(test = "x2"), cpdag = TRUE,
  debug = TRUE)
# * testing D - E for direction.
#   > testing D -> E
#     > bootstrap probability of an arc between D and E is 0.36 .
#     > direction confidence for arc D -> E is 0.3333333 .
#     > direction confidence for arc E -> D is 0.6666667 .
#   @ nothing to do, bootstrap probability is less than 0.50.

## End(Not run)
```

---

ci.test *Independence and Conditional Independence Tests*

Description

Perform either an independence test or a conditional independence test.

Usage

```
## S3 method for class 'character'
ci.test(x, y = NULL, z = NULL, data, test = NULL,
  B = NULL, debug = FALSE, ...)
## S3 method for class 'data.frame'
ci.test(x, test = NULL, B = NULL, debug = FALSE, ...)
## S3 method for class 'numeric'
ci.test(x, y = NULL, z = NULL, test = NULL,
  B = NULL, debug = FALSE, ...)
## S3 method for class 'factor'
ci.test(x, y = NULL, z = NULL, test = NULL,
  B = NULL, debug = FALSE, ...)
## Default S3 method:
ci.test(x, ...)
```

Arguments

x	a character string (the name of a variable), a data frame, a numeric vector or a factor object.
y	a character string (the name of another variable), a numeric vector or a factor object.
z	a vector of character strings (the names of the conditioning variables), a numeric vector, a factor object or a data frame. If NULL an independence test will be executed.

`data`	a data frame containing the variables to be tested.
`test`	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for discrete data sets and the *linear correlation* for continuous ones. See `bnlearn-package` for details.
`B`	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the `test` argument is not a permutation test.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
`...`	extra arguments from the generic method (currently ignored).

Value

An object of class `htest` containing the following components:

`statistic`	the value the test statistic.
`parameter`	the degrees of freedom of the approximate chi-squared or t distribution of the test statistic; the number of permutationscomputed by Monte Carlo tests. Semiparametric tests have both.
`p.value`	the p-value for the test.
`method`	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.
`data.name`	a character string giving the name(s) of the data.
`null.value`	the value of the test statistic under the null hypothesis, always 0.
`alternative`	a character string describing the alternative hypothesis

Author(s)

Marco Scutari

References

Edwards DI (2000). *Introduction to Graphical Modelling*. Springer, 2nd edition.

Hausser J, Strimmer K (2009). "Entropy inference and the James-Stein estimator, with application to nonlinear gene association networks". *Statistical Applications in Genetics and Molecular Biology*, **10**, 1469-1484.

Ledoit O, Wolf M (2003). "Improved Estimation of the Covariance Matrix of Stock Returns with an Application to Portfolio Selection". *Journal of Empirical Finance*, **10**, 603-621.

Legendre P (2000). "Comparison of Permutation Methods for the Partial Correlation and Partial Mantel Tests". *Journal of Statistical Computation and Simulation*, **67**, 37-73.

Tsamardinos I, Borboudakis G (2010). "Permutation Testing Improves Bayesian Network Learning". In "Machine Learning and Knowledge Discovery in Databases", pp. 322-337. Springer.

See Also

choose.direction, arc.strength.

Examples

```
data(gaussian.test)
data(learning.test)

# using a data frame and column labels.
ci.test(x = "F" , y = "B", z = c("C", "D"), data = gaussian.test)
#
#  Pearson's Linear Correlation
#
# data:  F ~ B | C + D
# cor = -0.1275, df = 4996, p-value < 2.2e-16
# alternative hypothesis: true value is not equal to 0

# using a data frame.
ci.test(gaussian.test)
#
#  Pearson's Linear Correlation
#
# data:  A ~ B | C + D + E + F + G
# cor = -0.5654, df = 4993, p-value < 2.2e-16
# alternative hypothesis: true value is not equal to 0

# using factor objects.
attach(learning.test)
ci.test(x = F , y = B, z = data.frame(C, D))
#
#  Mutual Information (discrete)
#
# data:  F ~ B | data.frame(C, D)
# mi = 25.2664, df = 18, p-value = 0.1178
# alternative hypothesis: true value is greater than 0
```

---

compare *Compare two different Bayesian networks*

Description

Compare two different Bayesian networks; compute the Structural Hamming Distance (SHD) between them or the Hamming distance between their skeletons.

Usage

```
compare(target, current, arcs = FALSE)
## S3 method for class 'bn'
all.equal(target, current, ...)
```

```
shd(learned, true, debug = FALSE)
hamming(learned, true, debug = FALSE)
```

Arguments

`target, learned`	an object of class bn.
`current, true`	another object of class bn.
`...`	extra arguments from the generic method (currently ignored).
`debug`	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.
`arcs`	a boolean value. See below.

Value

`compare` returns a list containing the number of true positives (`tp`, the number of arcs in `current` also present in `target`), of false positives (`fp`, the number of arcs in `current` not present in `target`) and of false negatives (`tn`, the number of arcs not in `current` but present in `target`) if `arcs` is `FALSE`; or the corresponding arc sets if `arcs` is `TRUE`.

`all.equal` returns either `TRUE` or a character string describing the differences between `target` and `current`.

`shd` and `hamming` return a non-negative integer number.

Author(s)

Marco Scutari

References

Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm". *Machine Learning*, **65**(1), 31-78.

Examples

```
data(learning.test)

e1 = model2network("[A][B][C|A:B][D|B][E|C][F|A:E]")
e2 = model2network("[A][B][C|A:B][D|B][E|C:F][F|A]")
shd(e2, e1, debug = TRUE)
# * arcs between A and F do not match.
#   > the learned network contains A - F.
#   > the true network contains A -> F.
# * arcs between E and F do not match.
#   > the learned network contains F -> E.
#   > the true network contains E -> F.
# [1] 2
unlist(compare(e1,e2))
# tp fp fn
```

```
#   5   1   1
compare(target = e1, current = e2, arcs = TRUE)
# $tp
#      from to
# [1,] "A"  "C"
# [2,] "B"  "C"
# [3,] "B"  "D"
# [4,] "C"  "E"
# [5,] "A"  "F"
#
# $fp
#      from to
# [1,] "F"  "E"
#
# $fn
#      from to
# [1,] "E"  "F"
```

---

constraint-based algorithms

Constraint-based structure learning algorithms

Description

Learn the equivalence class of a directed acyclic graph (DAG) from data using the Grow-Shrink (GS), the Incremental Association (IAMB), the Fast Incremental Association (Fast IAMB) or the Interleaved Incremental Association (Inter IAMB) constraint-based algorithms. Also use the same algorithms to learn the Markov blanket of a single variable.

Usage

```
gs(x, cluster = NULL, whitelist = NULL, blacklist = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE,
  optimized = TRUE, strict = FALSE, undirected = FALSE)
iamb(x, cluster = NULL, whitelist = NULL, blacklist = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE,
  optimized = TRUE, strict = FALSE, undirected = FALSE)
fast.iamb(x, cluster = NULL, whitelist = NULL, blacklist = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE,
  optimized = TRUE, strict = FALSE, undirected = FALSE)
inter.iamb(x, cluster = NULL, whitelist = NULL, blacklist = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE,
  optimized = TRUE, strict = FALSE, undirected = FALSE)
```

Arguments

x	a data frame containing the variables in the model.
cluster	an optional cluster object from package **snow**. See `snow integration` for details and a simple example.

`whitelist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`test`	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for discrete data sets and the *linear correlation* for continuous ones. See `bnlearn-package` for details.
`alpha`	a numeric value, the target nominal type I error rate.
`B`	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the `test` argument is not a permutation test.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
`optimized`	a boolean value. See `bnlearn-package` for details.
`strict`	a boolean value. If `TRUE` conflicting results in the learning process generate an error; otherwise they result in a warning.
`undirected`	a boolean value. If `TRUE` no attempt will be made to determine the orientation of the arcs; the returned (undirected) graph will represent the underlying structure of the Bayesian network.

Value

An object of class `bn`. See `bn-class` for details.

Author(s)

Marco Scutari

References

for Grow-Shrink (GS):

Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-03-153.

for Incremental Association (IAMB):

Tsamardinos I, Aliferis CF, Statnikov A (2003). "Algorithms for Large Scale Markov Blanket Discovery". In "Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference", pp. 376-381. AAAI Press.

for Fast IAMB and Inter IAMB:

Yaramakala S, Margaritis D (2005). "Speculative Markov Blanket Discovery for Optimal Feature Selection". In "ICDM '05: Proceedings of the Fifth IEEE International Conference on Data Mining", pp. 809-812. IEEE Computer Society.

See Also

local discovery algorithms, score-based algorithms, hybrid algorithms.

`coronary` *Coronary Heart Disease data set*

Description

Probable risk factors for coronary trombosis, comprising data from 1841 men.

Usage

```
data(coronary)
```

Format

The `coronary` data set contains the following 6 variables:

- `Smoking` (*smoking*): a two-level factor with levels `no` and `yes`.
- `M. Work` (*strenuous mental work*): a two-level factor with levels `no` and `yes`.
- `P. Work` (*strenuous physical work*): a two-level factor with levels `no` and `yes`.
- `Pressure` (*systolic blood pressure*): a two-level factor with levels `<140` and `>140`.
- `Proteins` (*ratio of beta and alpha lipoproteins*): a two-level factor with levels `<3` and `>3`.
- `Family` (*family anamnesis of coronary heart disease*): a two-level factor with levels `neg` and `pos`.

Source

Edwards DI (2000). *Introduction to Graphical Modelling*. Springer, 2nd edition.

Reinis Z, Pokorny J, Basika V, Tiserova J, Gorican K, Horakova D, Stuchlikova E, Havranek T, Hrabovsky F (1981). "Prognostic Significance of the Risk Profile in the Prevention of Coronary Heart Disease". Bratisl Lek Listy, **76**, 137-150. Published on Bratislava Medical Journal, in Czech.

Whittaker J (1990). *Graphical Models in Applied Multivariate Statistics*. Wiley.

Examples

```
# This is the undirected graphical model from Whittaker (1990).
data(coronary)
ug = empty.graph(names(coronary))
arcs(ug, ignore.cycles = TRUE) = matrix(
  c("Family", "M. Work", "M. Work", "Family",
    "M. Work", "P. Work", "P. Work", "M. Work",
    "M. Work", "Proteins", "Proteins", "M. Work",
    "M. Work", "Smoking", "Smoking", "M. Work",
    "P. Work", "Smoking", "Smoking", "P. Work",
```

```
    "P. Work", "Proteins", "Proteins", "P. Work",
    "Smoking", "Proteins", "Proteins", "Smoking",
    "Smoking", "Pressure", "Pressure", "Smoking",
    "Pressure", "Proteins", "Proteins", "Pressure"),
  ncol = 2, byrow = TRUE,
  dimnames = list(c(), c("from", "to")))
```

---

cpdag *Equivalence classes, moral graphs and consistent extenions*

Description

Find the equivalence class and the v-structures of a Bayesian network, construct its moral graph, or create a consistent extension of an equivalent class.

Usage

```
cpdag(x, moral = FALSE, debug = FALSE)
cextend(x, debug = FALSE)
vstructs(x, arcs = FALSE, moral = FALSE, debug = FALSE)
moral(x, debug = FALSE)
```

Arguments

`x`	an object of class `bn`.
`arcs`	a boolean value. If `TRUE` the arcs that are part of at least one v-structure are returned instead of the v-structures themselves.
`moral`	a boolean value. If `TRUE` we define a v-structure as in Pearl (2000); if `FALSE`, as in Koller and Friedman (2009). See below.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

What kind of arc configuration is called a v-structure is not uniquely defined in literature. The original definition from Pearl (2000), which is still followed by most texts and papers, states that the two parents in the v-structure must not be connected by an arc. However, Koller and Friedman (2009) call that a *immoral v-structure* and call a *moral v-structure* a v-structure in which the parents are linked by an arc. This mirrors the *unshielded* versus *shielded collider* naming convention, but is confusing.

Setting `moral` to `TRUE` in `cpdag` and `vstructs` makes those functions follow the definition from Koller and Friedman (2009); the default value of `FALSE`, on the other hand, makes those functions follow the definition from Pearl (2000).

Value

cpdag returns an object of class bn, representing the equivalence class. moral on the other hand returns the moral graph. See bn-class for details.

cextend returns an object of class bn, representing a DAG that is the consistent extension of x.

vstructs returns a matrix with either 2 or 3 columns, according to the value of the arcs parameter.

Author(s)

Marco Scutari

References

Dor D (1992). *A Simple Algorithm to Construct a Consistent Extension of a Partially Oriented Graph*. UCLA, Cognitive Systems Laboratory. Available as Technical Report R-185.

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Pearl J (2000). *Causality: Models, Reasoning and Inference*. Cambridge University Press.

Examples

```
data(learning.test)
res = gs(learning.test)
cpdag(res)
#
#   Bayesian network learned via Constraint-based methods
#
#   model:
#     [partially directed graph]
#   nodes:                                 6
#   arcs:                                  5
#     undirected arcs:                     1
#     directed arcs:                       4
#   average markov blanket size:           2.33
#   average neighbourhood size:            1.67
#   average branching factor:              0.67
#
#   learning algorithm:                    Grow-Shrink
#   conditional independence test:         Mutual Information (discrete)
#   alpha threshold:                       0.05
#   tests used in the learning procedure:  43
#   optimized:                             TRUE
#
vstructs(res)
#      X   Z   Y
# [1,] "A" "D" "C"
# [2,] "B" "E" "F"
```

cpquery *Perform conditional probability queries*

Description

Perform conditional probability queries (CPQs).

Usage

```
cpquery(fitted, event, evidence, cluster = NULL, method = "ls", ...,
  debug = FALSE)
cpdist(fitted, nodes, evidence, cluster = NULL, method = "ls", ...,
  debug = FALSE)
```

Arguments

`fitted`	an object of class `bn.fit`.
`event, evidence`	see below.
`nodes`	a vector of character strings, the labels of the nodes whose conditional distribution we are interested in.
`cluster`	an optional cluster object from package **snow**. See `snow integration` for details and a simple example.
`method`	a character string, the method used to perform the conditional probability query. Currently only *Logic Sampling* is implemented.
`...`	additional tuning parameters.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

`cpquery` returns a numeric value, the conditional probability of `event` conditional on `evidence`.

`cpudist` returns a data frame containing the observations generated from the conditional distribution of the `nodes` conditional on `evidence`.

Logic Sampling

The `event` and `evidence` arguments must be two expressions describing the event of interest and the conditioning evidence in a format such that, if we denote with `data` the data set the network was learned from, `data[evidence, ]` and `data[event, ]` return the correct observations. If either parameter is equal to `TRUE` an unconditional probability query is performed.

Two tuning parameters are available:

- `n`: a positive integer number, the number of random observations to generate from `fitted`. Defaults to `5000 * nparams(fitted)`.

- `batch`: a positive integer number, the size of each batch of random observations. Defaults to `10^4`.

Note that the number of observations returned by `cpdist` is always smaller than `n`, because logic sampling is a form of rejection sampling. Therefore, conly the obervations matching `evidence` (out of the `n` that are generated) are returned, and their number depends on the probability of `evidence`.

Author(s)

Marco Scutari

References

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.

Examples

```
## discrete Bayesian network.
fitted = bn.fit(hc(learning.test), learning.test)
# the result should be around 0.025.
cpquery(fitted, (B == "b"), (A == "a"))
# for a single observation, predict the value of a single
# variable conditional on the others.
var = names(learning.test)
obs = 2
str = paste("(", names(learning.test)[-3], "=='",
        sapply(learning.test[obs,-3], as.character), "')",
        sep = "", collapse = " & ")
str
str2 = paste("(", names(learning.test)[3], "=='",
         as.character(learning.test[obs, 3]), "')", sep = "")
str2
cpquery(fitted, eval(parse(text = str2)), eval(parse(text = str)))
# conditional distribution of A given C == "c".
table(cpdist(fitted, "A", (C == "c")))

## Gaussian Bayesian network.
fitted = bn.fit(hc(gaussian.test), gaussian.test)
# the result should be around 0.04.
cpquery(fitted,
  event = ((A >= 0) & (A <= 1)) & ((B >= 0) & (B <= 3)),
  evidence = (C + D < 10))
```

`deal integration` *bnlearn - deal package integration*

Description

How to use the **bnlearn** package with the Bayesian network learning methods provided by the **deal** package.

Export a bn object to deal

```
# load the bnlearn package.
> library(bnlearn)
> data(learning.test)
# learn the network structure.
> res = hc(learning.test)
> modelstring(res)
[1] "[A][C][F][B|A][D|A:C][E|B:F]"
# load the deal package.
> library(deal)

Attaching package: 'deal'


        The following object(s) are masked from package:bnlearn :

         modelstring,
         nodes,
         score

> bnlearn::node.ordering(res)
[1] "A" "C" "F" "B" "D" "E"
# create an empty network object.
> net = deal::network(learning.test[, node.ordering(res)])
# convert the bn object via its string representation.
> net = deal::as.network(bnlearn::modelstring(res), net)
# the network is the same, modulo some differences due to the
# partial ordering of the nodes.
> deal::modelstring(net)
[1] "[A][C][F][B|A][D|A:C][E|F:B]"
> bnlearn::modelstring(res)
[1] "[A][C][F][B|A][D|A:C][E|B:F]"
```

Import a network structure from deal

```
res2 = bnlearn::model2network(deal::modelstring(net))
```

Author(s)

Marco Scutari

`discretize` *Discretize data to learn discrete Bayesian networks*

Description

Discretize data to learn discrete Bayesian networks.

Usage

```
discretize(x, method, breaks = 3, ..., debug = FALSE)
```

Arguments

`x`	a data frame containing with numeric and factor columns.
`method`	a character string, either `interval` for *interval discretization*, `quantile` for *quantile discretization* (the default) or `hartemink` for *Hartemink's pairwise mutual information* method.
`breaks`	if `method` is set to `hartemink`, an integer number, the number of levels the variables are to be discretized into. Otherwise, a vector of integer numbers, one for each column of the data set, specifying the number of levels for each variable.
`...`	additional tuning parameters, see below.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

`discretize` returns a data frame with the same structure (number of columns, column names, etc.) as `x`, containing the discretized variables.

Note

Hartemink's algorithm has been designed to deal with sets of homogeneous, continuous variables; this is the reason why they are initially transformed into discrete variables, all with the same number of levels (given by the `ibreaks` argument). Which of the other algorithms is used is specified by the `idisc` argument (`quantile` is the default). The implementation in `bnlearn` also handles sets of discrete variables with the same number of levels, which are treated as adjacent interval identifiers. This allows the user to perform the initial discretization with the algorithm of his choice, as long as all variables have the same number of levels in the end.

Author(s)

Marco Scutari

References

Hartemink A (2001). *Principled Computational Methods for the Validation and Discovery of Genetic Regulatory Networks*. Ph.D. thesis, School of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Examples

```
data(gaussian.test)
d = discretize(gaussian.test, method = 'hartemink', breaks = 4, ibreaks = 20)
plot(hc(d))
```

`dsep` *Test d-separation*

Description

Check whether two nodes are d-separated.

Usage

```
dsep(bn, x, y, z)
```

Arguments

`bn`	an object of class `bn`.
`x,y`	a character string, the label of a node.
`z`	an optional vector of character strings, the label of the (candidate) d-separating nodes. It defaults to the empty set.

Value

`dsep` returns `TRUE` if `x` and `y` are d-separated by `z`, and `FALSE` otherwise.

Author(s)

Marco Scutari

References

Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Examples

```
bn = model2network("[A][C|A][B|C]")
dsep(bn, "A", "B", "C")
# [1] TRUE
bn = model2network("[A][C][B|A:C]")
dsep(bn, "A", "B", "C")
# [1] FALSE
```

---

foreign files utilities

Read and write BIF, NET and DSC files

Description

Read networks saved from other programs into `bn.fit` objects, and dump `bn.fit` objects into files for other programs to read.

Usage

```
# Old (non-XML) Bayesian Interchange files.
read.bif(file, debug = FALSE)
write.bif(file, fitted)

# Microsoft Interchange files.
read.dsc(file, debug = FALSE)
write.dsc(file, fitted)

# HUGIN flat network format.
read.net(file, debug = FALSE)
write.net(file, fitted)
```

Arguments

`file`	a connection object or a character string.
`fitted`	an object of class `bn.fit`.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

`read.bif`, `read.dsc` and `read.net` return an object of class `bn.fit`.

`write.bif`, `write.dsc` and `write.net` return `NULL` invisibly.

Note

Most of the networks present in the Bayesian Network Repository have an associated BIF file that can be imported with `read.bif`.

HUGIN can import and export NET files; Netica can read (but not write) DSC files; and Genie can read and write both DSC and NET files.

Please note that these functions work on a "best effort" basis, as the parsing of these formats have been implementing by reverse engineering the file format from publicly available examples.

Author(s)

Marco Scutari

References

Elidan G (2001). "Bayesian Network Repository". http://www.cs.huji.ac.il/site/labs/compbio/Repository.

Genie, http://genie.sis.pitt.edu.

HUGIN Expert, http://www.hugin.com.

Netica, http://www.norsys.com/netica.html.

`gaussian.test` *Synthetic (continuous) data set to test learning algorithms*

Description

This a synthetic data set used as a test case in the **bnlearn** package.

Usage

```
data(gaussian.test)
```

Format

The `gaussian.test` data set contains the seven normal (Gaussian) variables.

Note

The R script to generate data from this network is shipped in the ‘`network.scripts`’ directory of this package.

Examples

```
# load the data and build the correct network from the model string.
data(gaussian.test)
res = empty.graph(names(gaussian.test))
modelstring(res) = "[A][B][E][G][C|A:B][D|B][F|A:D:E:G]"
plot(res)
```

gRain integration *Import and export networks from the gRain package*

Description

Convert bn.fit objects to grain objects and vice versa.

Usage

```
## S3 method for class 'grain'
as.bn.fit(x)
## S3 method for class 'bn.fit'
as.grain(x)
```

Arguments

x	an object of class grain (for as.bn.fit) or bn.fit (for as.grain).

Value

An object of class grain (for as.grain) or bn.fit (for as.bn.fit).

Note

Conditional probability tables in grain objects must be completely specified; on the other hand, bn.fit allows NaN values for unobserved parents' configurations. Such bn.fit objects cannot be converted to grain objects as they are. A simple solution to this problem is to fit another bn.fit with method = "bayes" and a low iss value, using the same data and network structure.

Author(s)

Marco Scutari

Examples

```
## Not run:
library(gRain)
a = bn.fit(hc(learning.test), learning.test)
b = as.grain(a)
c = as.bn.fit(b)
## End(Not run)
```

`graph generation utilities` *Generate empty or random graphs*

Description

Generate empty or random directed acyclic graphs from a given set of nodes.

Usage

```
empty.graph(nodes, num = 1)
random.graph(nodes, num = 1, method = "ordered", ...,
  debug = FALSE)
```

Arguments

`nodes`	a vector of character strings, the labels of the nodes.
`num`	an integer, the number of graphs to be generated.
`method`	a character string, the label of a score. Possible values are `ordered` (*full ordering* based generation), `ic-dag` (Ide's and Cozman's *Generating Multi-connected DAGs* algorithm), `melancon` (Melancon's and Philippe's *Uniform Random Acyclic Digraphs* algorithm) and `empty` (generates empty graphs).
`...`	additional tuning parameters (see below).
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent. Ignored in some generation methods.

Details

Available graph generation algorithms are:

- *full ordering* based generation (`ordered`): generates graphs whose node ordering is given by the order of the labels in the `nodes` parameter. The same algorithm is used in the `randomDAG` function in package **pcalg**.
- Ide's and Cozman's *Generating Multi-connected DAGs* algorithm (`ic-dag`): generates graphs with a uniform probability distribution over the set of multiconnected graphs.
- Melancon's and Philippe's *Uniform Random Acyclic Digraphs* algorithm (`melancon`): generates graphs with a uniform probability distribution over the set of all possible graphs.
- *empty graphs* (`empty`): generates graphs without any arc.

Additional parameters for the `random.graph` function are:

- `prob`: the probability of each arc to be present in a graph generated by the `ordered` algorithm. The default value is `2 / (length(nodes) - 1)`, which results in a sparse graph (the number of arcs should be of the same order as the number of nodes).
- `burn.in`: the number of iterations for the `ic-dag` and `melancon` algorithms to converge to a stationary (and uniform) probability distribution. The default value is `6 * length(nodes)^2`.

- every: return only one graph every number of steps instead of all the graphs generated with ic-dag and melancon. Since both algorithms are based on Markov Chain Monte Carlo approaches, high values of every result in a more diverse set of networks. The default value is 1, i.e. to return all the networks that are generated.
- max.degree: the maximum degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.
- max.in.degree: the maximum in-degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.
- max.out.degree: the maximum out-degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.

Value

Both empty.graph and random.graph return an object of class bn (if num is equal to 1) or a list of objects of class bn (otherwise). If every is greated than 1, random.graph always returns a list, regardless of the number of graphs it contains.

Author(s)

Marco Scutari

References

Ide JS, Cozman FG (2002). "Random Generation of Bayesian Networks". In "SBIA '02: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence", pp. 366-375. Springer-Verlag.

Melancon G, Dutour I, Bousquet-Melou M (2001). "Random Generation of Directed Acyclic Graphs". *Electronic Notes in Discrete Mathematics*, **10**, 202-207.

Melancon G, Philippe F (2004). "Generating Connected Acyclic Digraphs Uniformly at Random". *Information Processing Letters*, **90**(4), 209-213.

Examples

```
empty.graph(LETTERS[1:8])
#
#  Random/Generated Bayesian network
#
#  model:
#    [A][B][C][D][E][F][G][H]
#  nodes:                            8
#  arcs:                             0
#    undirected arcs:                0
#    directed arcs:                  0
#  average markov blanket size:      0.00
#  average neighbourhood size:       0.00
#  average branching factor:         0.00
#
#  generation algorithm:             Empty
#
random.graph(LETTERS[1:8])
```

```
# <insert the description of a random graph here>
plot(random.graph(LETTERS[1:8], method = "ic-dag", max.in.degree = 2))
plot(random.graph(LETTERS[1:8]))
plot(random.graph(LETTERS[1:8], prob = 0.2))
```

---

graph integration *Import and export networks from the graph package*

Description

Convert bn and bn.fit objects to graphNEL and graphAM objects and vice versa.

Usage

```
## S3 method for class 'graphNEL'
as.bn(x)
## S3 method for class 'graphAM'
as.bn(x)
## S3 method for class 'bn'
as.graphNEL(x)
## S3 method for class 'bn.fit'
as.graphNEL(x)
## S3 method for class 'bn'
as.graphAM(x)
## S3 method for class 'bn.fit'
as.graphAM(x)
```

Arguments

x	an object of class bn, bn.fit, graphNEL, graphAM.

Value

An object of the relevant class.

Note

The corresponding S4 methods are exported as well, and are just wrappers around the S3 ones. So, for example, both `as.graphNEL(x)` and `as(x, "graphNEL")` work and return identical objects.

Author(s)

Marco Scutari

Examples

```
## Not run:
library(graph)
a = bn.fit(hc(learning.test), learning.test)
b = as.graphNEL(a)
c = as.bn(b)
identical(as.graphNEL(a), as(a, "graphNEL"))
## End(Not run)
```

`graph utilities` *Utilities to manipulate graphs*

Description

Check and manipulate graph-related properties of an object of class bn.

Usage

```
# check whether the graph is acyclic/completely directed.
acyclic(x, debug = FALSE)
directed(x)
# check whether there is a path between two nodes.
path(x, from, to, direct = TRUE, underlying.graph = FALSE,
  debug = FALSE)
# build the skeleton or a complete orientation of the graph.
skeleton(x)
pdag2dag(x, ordering)
# build a subgraph spanning a subset of nodes.
subgraph(x, nodes)
```

Arguments

`x`	an object of class bn. `acyclic`, `directed` and `path` also accept objects of class `bn.fit`.
`from`	a character string, the label of a node.
`to`	a character string, the label of a node (different from `from`).
`direct`	a boolean value. If `FALSE` ignore any arc between `from` and `to` when looking for a path.
`underlying.graph`	a boolean value. If `TRUE` the underlying undirected graph is used instead of the (directed) one from the `x` parameter.
`ordering`	the labels of all the nodes in the graph; their order is the node ordering used to set the direction of undirected arcs.
`nodes`	the labels of the nodes that induce the subgraph.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

acyclic, path and directed return a boolean value.
skeleton, pdag2dag and subgraph return an object of class bn.

Author(s)

Marco Scutari

References

Bang-Jensen J, Gutin G (2009). *Digraphs: Theory, Algorithms and Applications*. Springer, 2nd edition.

Examples

```
data(learning.test)
res = gs(learning.test)

acyclic(res)
# [1] TRUE
directed(res)
# [1] FALSE
res = pdag2dag(res, ordering = LETTERS[1:6])
res
#
#   Bayesian network learned via Constraint-based methods
#
#   model:
#     [A][C][F][B|A][D|A:C][E|B:F]
#   nodes:                                 6
#   arcs:                                  5
#     undirected arcs:                     0
#     directed arcs:                       5
#   average markov blanket size:           2.33
#   average neighbourhood size:            1.67
#   average branching factor:              0.83
#
#   learning algorithm:                    Grow-Shrink
#   conditional independence test:         Mutual Information (discrete)
#   alpha threshold:                       0.05
#   tests used in the learning procedure:  43
#   optimized:                             TRUE
#
directed(res)
# [1] TRUE
skeleton(res)
#
#   Bayesian network learned via Constraint-based methods
#
#   model:
#     [partially directed graph]
#   nodes:                                 6
```

```
#   arcs:                                  5
#     undirected arcs:                     5
#     directed arcs:                       0
#   average markov blanket size:           1.67
#   average neighbourhood size:            1.67
#   average branching factor:              0.00
#
#   learning algorithm:                    Grow-Shrink
#   conditional independence test:         Mutual Information (discrete)
#   alpha threshold:                       0.05
#   tests used in the learning procedure:  43
#   optimized:                             TRUE
#
```

`graphviz.plot` *Advanced Bayesian network plots*

Description

Plot the graph associated with a Bayesian network using the **Rgraphviz** package.

Usage

```
graphviz.plot(x, highlight = NULL, layout = "dot",
  shape = "circle", main = NULL, sub = NULL)
```

Arguments

`x`	an object of class `bn` or `bn.fit`.
`highlight`	a list, see below.
`layout`	a character string, the layout parameter to be passed to **Rgraphviz**. Possible values are `dots`, `neato`, `twopi`, `circo` and `fdp`. See **Rgraphviz** documentation for details.
`shape`	a character string, the shape of the nodes. Can be either `circle` or `ellipse`.
`main`	a character string, the main title of the graph. It's plotted at the top of the graph.
`sub`	a character string, a subtitle which is plotted at the bottom of the graph.

Details

The highlight parameter is a list with at least one of the following elements:

- `nodes`: a character vector, the labels of the nodes to be highlighted.
- `arcs`: the arcs to be highlighted (a two-column matrix, whose columns are labeled `from` and `to`).

and optionally one or more of the following formatting parameters:

- `col`: an integer or character string (the highlight colour for the arcs and the node frames). The default value is `red`.
- `textCol`: an integer or character string (the highlight colour for the labels of the nodes). The default value is `black`.
- `fill`: an integer or character string (the colour used as a background colour for the nodes). The default value is `white`.
- `lwd`: a positive number (the line width of highlighted arcs). It overrides the line width settings in `strength.plot`. The default value is to use the global settings of **Rgraphviz**.
- `lty`: the line type of highlighted arcs. Possible values are 0, 1, 2, 3, 4, 5, 6, "blank", "solid", "dashed", "dotted", "dotdash", "longdash" and "twodash". The default value is to use the global settings of **Rgraphviz**.

Value

`graphviz.plot` returns invisibly the `graph` object produced by **Rgraphviz**.

Author(s)

Marco Scutari

See Also

`plot.bn`.

`hailfinder` *The HailFinder weather forecast system (synthetic) data set*

Description

Hailfinder is a Bayesian network designed to forecast severe summer hail in northeastern Colorado.

Usage

```
data(hailfinder)
```

Format

The `hailfinder` data set contains the following 56 variables:

- `N07muVerMo` (*10.7mu vertical motion*): a four-level factor with levels `StrongUp`, `WeakUp`, `Neutral` and `Down`.
- `SubjVertMo` (*subjective judgment of vertical motion*): a four-level factor with levels `StrongUp`, `WeakUp`, `Neutral` and `Down`.
- `QGVertMotion` (*quasigeostrophic vertical motion*): a four-level factor with levels `StrongUp`, `WeakUp`, `Neutral` and `Down`.

- `CombVerMo` (*combined vertical motion*): a four-level factor with levels `StrongUp`, `WeakUp`, `Neutral` and `Down`.
- `AreaMesoALS` (*area of meso-alpha*): a four-level factor with levels `StrongUp`, `WeakUp`, `Neutral` and `Down`.
- `SatContMoist` (*satellite contribution to moisture*): a four-level factor with levels `VeryWet`, `Wet`, `Neutral` and `Dry`.
- `RaoContMoist` (*reading at the forecast center for moisture*): a four-level factor with levels `VeryWet`, `Wet`, `Neutral` and `Dry`.
- `CombMoisture` (*combined moisture*): a four-level factor with levels `VeryWet`, `Wet`, `Neutral` and `Dry`.
- `AreaMoDryAir` (*area of moisture and adry air*): a four-level factor with levels `VeryWet`, `Wet`, `Neutral` and `Dry`.
- `VISCloudCov` (*visible cloud cover*): a three-level factor with levels `Cloudy`, `PC` and `Clear`.
- `IRCloudCover` (*infrared cloud cover*): a three-level factor with levels `Cloudy`, `PC` and `Clear`.
- `CombClouds` (*combined cloud cover*): a three-level factor with levels `Cloudy`, `PC` and `Clear`.
- `CldShadeOth` (*cloud shading, other*): a three-level factor with levels `Cloudy`, `PC` and `Clear`.
- `AMInstabMt` (*AM instability in the mountains*): a three-level factor with levels `None`, `Weak` and `Strong`.
- `InsInMt` (*instability in the mountains*): a three-level factor with levels `None`, `Weak` and `Strong`.
- `WndHodograph` (*wind hodograph*): a four-level factor with levels `DCVZFavor`, `StrongWest`, `Westerly` and `Other`.
- `OutflowFrMt` (*outflow from mountains*): a three-level factor with levels `None`, `Weak` and `Strong`.
- `MorningBound` (*morning boundaries*): a three-level factor with levels `None`, `Weak` and `Strong`.
- `Boundaries` (*boundaries*): a three-level factor with levels `None`, `Weak` and `Strong`.
- `CldShadeConv` (*cloud shading, convection*): a three-level factor with levels `None`, `Some` and `Marked`.
- `CompPlFcst` (*composite plains forecast*): a three-level factor with levels `IncCapDecIns`, `LittleChange` and `DecCapIncIns`.
- `CapChange` (*capping change*): a three-level factor with levels `Decreasing`, `LittleChange` and `Increasing`.
- `LoLevMoistAd` (*low-level moisture advection*): a four-level factor with levels `StrongPos`, `WeakPos`, `Neutral` and `Negative`.
- `InsChange` (*instability change*): three-level factor with levels `Decreasing`, `LittleChange` and `Increasing`.
- `MountainFcst` (*mountains (region 1) forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `Date` (*date*): a six-level factor with levels `May15_Jun14`, `Jun15_Jul1`, `Jul2_Jul15`, `Jul16_Aug10`, `Aug11_Aug20` and `Aug20_Sep15`.
- `Scenario` (*scenario*): an eleven-level factor with levels `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J` and `K`.
- `ScenRelAMCIN` (*scenario relevant to AM convective inhibition*): a two-level factor with levels `AB` and `CThruK`.

- `MorningCIN` (*morning convective inhibition*): a four-level factor with levels `None`, `PartInhibit`, `Stifling` and `TotalInhibit`.
- `AMCINInScen` (*AM convective inhibition in scenario*): a three-level factor with levels `LessThanAve`, `Average` and `MoreThanAve`.
- `CapInScen` (*capping withing scenario*): a three-level factor with levels `LessThanAve`, `Average` and `MoreThanAve`.
- `ScenRelAMIns` (*scenario relevant to AM instability*): a six-level factor with levels `ABI`, `CDEJ`, `F`, `G`, `H` and `K`.
- `LIfr12ZDENSd` (*LI from 12Z DEN sounding*): a four-level factor with levels `LIGt0`, `N1GtLIGt_4`, `N5GtLIGt_8` and `LILt_8`.
- `AMDewptCalPl` (*AM dewpoint calculations, plains*): a three-level factor with levels `Instability`, `Neutral` and `Stability`.
- `AMInsWliScen` (*AM instability within scenario*): a three-level factor with levels `LessUnstable`, `Average` and `MoreUnstable`.
- `InsSclInScen` (*instability scaling within scenario*): a three-level factor with levels `LessUnstable`, `Average` and `MoreUnstable`.
- `ScenRel34` (*scenario relevant to regions 2/3/4*): a five-level factor with levels `ACEFK`, `B`, `D`, `GJ` and `HI`.
- `LatestCIN` (*latest convective inhibition*): a four-level factor with levels `None`, `PartInhibit`, `Stifling` and `TotalInhibit`.
- `LLIW` (*LLIW severe weather index*): a four-level factor with levels `Unfavorable`, `Weak`, `Moderate` and `Strong`.
- `CurPropConv` (*current propensity to convection*): a four-level factor with levels `None`, `Slight`, `Moderate` and `Strong`.
- `ScnRelPlFcst` (*scenario relevant to plains forecast*): an eleven-level factor with levels `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J` and `K`.
- `PlainsFcst` (*plains forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `N34StarFcst` (*regions 2/3/4 forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `R5Fcst` (*region 5 forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `Dewpoints` (*dewpoints*): a seven-level factor with levels `LowEverywhere`, `LowAtStation`, `LowSHighN`, `LowNHighS`, `LowMtsHighPl`, `HighEverywher`, `Other`.
- `LowLLapse` (*low-level lapse rate*): a four-level factor with levels `CloseToDryAd`, `Steep`, `ModerateOrLe` and `Stable`.
- `MeanRH` (*mean relative humidity*): a three-level factor with levels `VeryMoist`, `Average` and `Dry`.
- `MidLLapse` (*mid-level lapse rate*): a three-level factor with levels `CloseToDryAd`, `Steep` and `ModerateOrLe`.
- `MvmtFeatures` (*movement of features*): a four-level factor with levels `StrongFront`, `MarkedUpper`, `OtherRapid` and `NoMajor`.
- `RHRatio` (*realtive humidity ratio*): a three-level factor with levels `MoistMDryL`, `DryMMoistL` and `other`.

- SfcWndShfDis (*surface wind shifts and discontinuities*): a seven-level factor with levels DenvCyclone, E_W_N, E_W_S, MovigFtorOt, DryLine, None and Other.
- SynForcng (*synoptic forcing*): a five-level factor with levels SigNegative, NegToPos, SigPositive, PosToNeg and LittleChange.
- TempDis (*temperature discontinuities*): a four-level factor with levels QStationary, Moving, None, Other.
- WindAloft (*wind aloft*): a four-level factor with levels LV, SWQuad, NWQuad, AllElse.
- WindFieldMt (*wind fields, mountains*): a two-level factor with levels Westerly and LVorOther.
- WindFieldPln (*wind fields, plains*): a six-level factor with levels LV, DenvCyclone, LongAnticyc, E_NE, SEquad and WidespdDnsl.

Note

The R script to generate data from this network is shipped in the 'network.scripts' directory of this package.

Source

Abramson B, Brown J, Edwards W, Murphy A, Winkler RL (1996). "Hailfinder: A Bayesian system for forecasting severe weather". *International Journal of Forecasting*, **12**(1), 57-71.

Elidan G (2001). "Bayesian Network Repository".
http://www.cs.huji.ac.il/site/labs/compbio/Repository.

Examples

```
# load the data and build the correct network from the model string.
data(hailfinder)
res = empty.graph(names(hailfinder))
modelstring(res) = paste("[N07muVerMo][SubjVertMo][QGVertMotion]",
  "[SatContMoist][RaoContMoist][VISCloudCov][IRCloudCover][AMInstabMt]",
  "[WndHodograph][MorningBound][LoLevMoistAd][Date][MorningCIN]",
  "[LIfr12ZDENSd][AMDewptCalPl][LatestCIN][LLIW]",
  "[CombVerMo|N07muVerMo:SubjVertMo:QGVertMotion]",
  "[CombMoisture|SatContMoist:RaoContMoist]",
  "[CombClouds|VISCloudCov:IRCloudCover][Scenario|Date]",
  "[CurPropConv|LatestCIN:LLIW][AreaMesoALS|CombVerMo]",
  "[ScenRelAMCIN|Scenario][ScenRelAMIns|Scenario][ScenRel34|Scenario]",
  "[ScnRelPlFcst|Scenario][Dewpoints|Scenario][LowLLapse|Scenario]",
  "[MeanRH|Scenario][MidLLapse|Scenario][MvmtFeatures|Scenario]",
  "[RHRatio|Scenario][SfcWndShfDis|Scenario][SynForcng|Scenario]",
  "[TempDis|Scenario][WindAloft|Scenario][WindFieldMt|Scenario]",
  "[WindFieldPln|Scenario][AreaMoDryAir|AreaMesoALS:CombMoisture]",
  "[AMCINInScen|ScenRelAMCIN:MorningCIN]",
  "[AMInsWliScen|ScenRelAMIns:LIfr12ZDENSd:AMDewptCalPl]",
  "[CldShadeOth|AreaMesoALS:AreaMoDryAir:CombClouds]",
  "[InsInMt|CldShadeOth:AMInstabMt][OutflowFrMt|InsInMt:WndHodograph]",
  "[CldShadeConv|InsInMt:WndHodograph][MountainFcst|InsInMt]",
  "[Boundaries|WndHodograph:OutflowFrMt:MorningBound]",
  "[CompPlFcst|AreaMesoALS:CldShadeOth:Boundaries:CldShadeConv]",
```

```
  "[CapChange|CompPlFcst][InsChange|CompPlFcst:LoLevMoistAd]",
  "[CapInScen|CapChange:AMCINInScen]",
  "[InsSclInScen|InsChange:AMInsWliScen]",
  "[PlainsFcst|CapInScen:InsSclInScen:CurPropConv:ScnRelPlFcst]",
  "[N34StarFcst|ScenRel34:PlainsFcst][R5Fcst|MountainFcst:N34StarFcst]",
  sep = "")
## Not run:
# there are too many nodes for plot(), use graphviz.plot().
graphviz.plot(res)
## End(Not run)
```

`hybrid algorithms` *Hybrid structure learning algorithms*

Description

Learn the structure of a Bayesian network with the Max-Min Hill Climbing (MMHC) and the more general 2-phase Restricted Maximization (RSMAX2) hybrid algorithms.

Usage

```
rsmax2(x, whitelist = NULL, blacklist = NULL, restrict,
  maximize = "hc", test = NULL, score = NULL, alpha = 0.05,
  B = NULL, ..., maximize.args = list(), optimized = TRUE,
  strict = FALSE, debug = FALSE)
mmhc(x, whitelist = NULL, blacklist = NULL, test = NULL,
  score = NULL, alpha = 0.05, B = NULL, ..., restart = 0,
  perturb = 1, max.iter = Inf, optimized = TRUE,
  strict = FALSE, debug = FALSE)
```

Arguments

`x`	a data frame containing the variables in the model.
`whitelist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`restrict`	a character string, the constraint-based algorithm to be used in the "restrict" phase. Possible values are `gs`, `iamb`, `fast.iamb`, `inter.iamb` and `mmpc`. See `bnlearn-package` and the documentation of each algorithm for details.
`maximize`	a character string, the score-based algorithm to be used in the "maximize" phase. Possible values are `hc` and `tabu`. See `bnlearn-package` for details.
`test`	a character string, the label of the conditional independence test to be used by the constraint-based algorithm. If none is specified, the default test statistic is the *mutual information* for discrete data sets and the *linear correlation* for continuous ones. See `bnlearn-package` for details.

score	a character string, the label of the network score to be used in the score-based algorithm. If none is specified, the default score is the *Bayesian Information Criterion* for both discrete and continuous data sets. See bnlearn-package for details.
alpha	a numeric value, the target nominal type I error rate of the conditional independence test.
B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the test argument is not a permutation test.
...	additional tuning parameters for the network score used by the score-based algorithm. See score for details.
maximize.args	a list of arguments to be passed to the score-based algorithm specified by maximize, such as restart for hill-climbing or tabu for tabu search.
restart	an integer, the number of random restarts for the score-based algorithm.
perturb	an integer, the number of attempts to randomly insert/remove/reverse an arc on every random restart.
max.iter	an integer, the maximum number of iterations for the score-based algorithm.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.
optimized	a boolean value. See bnlearn-package for details.
strict	a boolean value. If TRUE conflicting results in the learning process generate an error; otherwise they result in a warning.

Value

An object of class bn. See bn-class for details.

Note

mmhc is simply rshc with restrict set to mmpc and maximize set to hc.

Author(s)

Marco Scutari

References

Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm". *Machine Learning*, **65**(1), 31-78.

See Also

local discovery algorithms, score-based algorithms, constraint-based algorithms.

`insurance` *Insurance evaluation network (synthetic) data set*

Description

Insurance is a network for evaluating car insurance risks.

Usage

```
data(insurance)
```

Format

The `insurance` data set contains the following 27 variables:

- `GoodStudent` (*good student*): a two-level factor with levels `False` and `True`.
- `Age` (*age*): a three-level factor with levels `Adolescent`, `Adult` and `Senior`.
- `SocioEcon` (*socio-economic status*): a four-level factor with levels `Prole`, `Middle`, `UpperMiddle` and `Wealthy`.
- `RiskAversion` (*risk aversion*): a four-level factor with levels `Psychopath`, `Adventurous`, `Normal` and `Cautious`.
- `VehicleYear` (*vehicle age*): a two-level factor with levels `Current` and `older`.
- `ThisCarDam` (*damage to this car*): a four-level factor with levels `None`, `Mild`, `Moderate` and `Severe`.
- `RuggedAuto` (*ruggedness of the car*): a three-level factor with levels `EggShell`, `Football` and `Tank`.
- `Accident` (*severity of the accident*): a four-level factor with levels `None`, `Mild`, `Moderate` and `Severe`.
- `MakeModel` (*car's model*): a five-level factor with levels `SportsCar`, `Economy`, `FamilySedan`, `Luxury` and `SuperLuxury`.
- `DrivQuality` (*driving quality*): a three-level factor with levels `Poor`, `Normal` and `Excellent`.
- `Mileage` (*mileage*): a four-level factor with levels `FiveThou`, `TwentyThou`, `FiftyThou` and `Domino`.
- `Antilock` (*ABS*): a two-level factor with levels `False` and `True`.
- `DrivingSkill` (*driving skill*): a three-level factor with levels `SubStandard`, `Normal` and `Expert`.
- `SeniorTrain` (*senior training*): a two-level factor with levels `False` and `True`.
- `ThisCarCost` (*costs for the insured car*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `Theft` (*theft*): a two-level factor with levels `False` and `True`.
- `CarValue` (*value of the car*): a five-level factor with levels `FiveThou`, `TenThou`, `TwentyThou`, `FiftyThou` and `Million`.

- HomeBase (*neighbourhood type*): a four-level factor with levels Secure, City, Suburb and Rural.
- AntiTheft (*anti-theft system*): a two-level factor with levels False and True.
- PropCost (*ratio of the cost for the two cars*): a four-level factor with levels Thousand, TenThou, HundredThou and Million.
- OtherCarCost (*costs for the other car*): a four-level factor with levels Thousand, TenThou, HundredThou and Million.
- OtherCar (*other cars involved in the accident*): a two-level factor with levels False and True.
- MedCost (*cost of the medical treatment*): a four-level factor with levels Thousand, TenThou, HundredThou and Million.
- Cushioning (*cushioning*): a four-level factor with levels Poor, Fair, Good and Excellent.
- Airbag (*airbag*): a two-level factor with levels False and True.
- ILiCost (*inspection cost*): a four-level factor with levels Thousand, TenThou, HundredThou and Million.
- DrivHist (*driving history*): a three-level factor with levels Zero, One and Many.

Note

The R script to generate data from this network is shipped in the 'network.scripts' directory of this package.

Source

Binder J, Koller D, Russell S, Kanazawa K (1997). "Adaptive Probabilistic Networks with Hidden Variables". *Machine Learning*, **29**(2-3), 213-244.

Elidan G (2001). "Bayesian Network Repository".
http://www.cs.huji.ac.il/site/labs/compbio/Repository.

Examples

```
# load the data and build the correct network from the model string.
data(insurance)
res = empty.graph(names(insurance))
modelstring(res) = paste("[Age][Mileage][SocioEcon|Age]",
  "[GoodStudent|Age:SocioEcon][RiskAversion|Age:SocioEcon]",
  "[OtherCar|SocioEcon][VehicleYear|SocioEcon:RiskAversion]",
  "[MakeModel|SocioEcon:RiskAversion][SeniorTrain|Age:RiskAversion]",
  "[HomeBase|SocioEcon:RiskAversion][AntiTheft|SocioEcon:RiskAversion]",
  "[RuggedAuto|VehicleYear:MakeModel][Antilock|VehicleYear:MakeModel]",
  "[DrivingSkill|Age:SeniorTrain][CarValue|VehicleYear:MakeModel:Mileage]",
  "[Airbag|VehicleYear:MakeModel][DrivQuality|RiskAversion:DrivingSkill]",
  "[Theft|CarValue:HomeBase:AntiTheft][Cushioning|RuggedAuto:Airbag]",
  "[DrivHist|RiskAversion:DrivingSkill]",
  "[Accident|DrivQuality:Mileage:Antilock]",
  "[ThisCarDam|RuggedAuto:Accident][OtherCarCost|RuggedAuto:Accident]",
  "[MedCost|Age:Accident:Cushioning][ILiCost|Accident]",
  "[ThisCarCost|ThisCarDam:Theft:CarValue]",
```

```
  "[PropCost|ThisCarCost:OtherCarCost]", sep = "")
## Not run:
# there are too many nodes for plot(), use graphviz.plot().
graphviz.plot(res)
## End(Not run)
```

`learning.test` *Synthetic (discrete) data set to test learning algorithms*

Description

This a synthetic data set used as a test case in the **bnlearn** package.

Usage

```
data(learning.test)
```

Format

The `learning.test` data set contains the following variables:

- `A`, a three-level factor with levels `a`, `b` and `c`.
- `B`, a three-level factor with levels `a`, `b` and `c`.
- `C`, a three-level factor with levels `a`, `b` and `c`.
- `D`, a three-level factor with levels `a`, `b` and `c`.
- `E`, a three-level factor with levels `a`, `b` and `c`.
- `F`, a two-level factor with levels `a` and `b`.

Note

The R script to generate data from this network is shipped in the ‘`network.scripts`’ directory of this package.

Examples

```
# load the data and build the correct network from the model string.
data(learning.test)
res = empty.graph(names(learning.test))
modelstring(res) = "[A][C][F][B|A][D|A:C][E|B:F]"
plot(res)
```

`lizards` *Lizards' perching behaviour data set*

Description

Real-world data set about the perching behaviour of two species of lizards in the South Bimini island, from Shoener (1968).

Usage

```
data(lizards)
```

Format

The `lizards` data set contains the following variables:

- `Species` (*the species of the lizard*): a two-level factor with levels `Sagrei` and `Distichus`.
- `Height` (*perch height*): a two-level factor with levels `high` (greater than 4.75 feet) and `low` (lesser or equal to 4.75 feet).
- `Diameter` (*perch diameter*): a two-level factor with levels `narrow` (greater than 4 inches) and `wide` (lesser or equal to 4 inches).

Source

Edwards DI (2000). *Introduction to Graphical Modelling*. Springer, 2nd edition.

Fienberg SE (1980). *The Analysis of Cross-Classified Categorical Data*. Springer, 2nd edition.

Schoener TW (1968). "The Anolis Lizards of Bimini: Resource Partitioning in a Complex Fauna". *Ecology*, **49**(4), 704-726.

Examples

```
# load the data and build the correct network from the model string.
data(lizards)
res = empty.graph(names(lizards))
modelstring(res) = "[Species][Diameter|Species][Height|Species]"
plot(res)

table(lizards[, c(3,2,1)])
# , , Species = Sagrei
#
#       Diameter
# Height narrow wide
#   high     86   35
#   low      32   11
#
# , , Species = Distichus
#
```

```
#       Diameter
# Height narrow wide
#   high     73   70
#   low      61   41

## Not run:
# This data set is useful as it offers nominal values for
# the conditional mutual information and X^2 tests.

attach(lizards)
ci.test(Height, Diameter, Species, test = "mi")
#
#  Mutual Information (discrete)
#
# data:  Height ~ Diameter | Species
# mi = 2.0256, df = 2, p-value = 0.3632
# alternative hypothesis: true value is greater than 0
ci.test(Height, Diameter, Species, test = "x2")
#
#  Pearson's X^2
#
# data:  Height ~ Diameter | Species
# x2 = 2.0174, df = 2, p-value = 0.3647
# alternative hypothesis: true value is greater than 0

## End(Not run)
```

`local discovery algorithms`
Local discovery structure learning algorithms

Description

Learn the skeleton of a directed acyclic graph (DAG) from data using the Max-Min Parents and Children (MMPC) and the Semi-Interleaved HITON-PC constraint-based algorithms. ARACNE and Chow-Liu learn an approximation of that structure using pairwise mutual information coefficients.

Usage

```
mmpc(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, debug = FALSE, optimized = TRUE, strict = FALSE)
si.hiton.pc(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, debug = FALSE, optimized = TRUE, strict = FALSE)

aracne(x, whitelist = NULL, blacklist = NULL, mi = NULL, debug = FALSE)
chow.liu(x, whitelist = NULL, blacklist = NULL, mi = NULL, debug = FALSE)
```

Arguments

- `x` a data frame containing the variables in the model.
- `cluster` an optional cluster object from package **snow**. See `snow integration` for details and a simple example.
- `whitelist` a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
- `blacklist` a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
- `mi` a character string, the estimator used for the pairwise (i.e. unconditional) mutual information coefficients in the ARACNE and Chow-Liu algorithms. Possible values are `mi` (discrete mutual information) and `mi-g` (Gaussian mutual information).
- `test` a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for discrete data sets and the *linear correlation* for continuous ones. See `bnlearn-package` for details.
- `alpha` a numeric value, the target nominal type I error rate.
- `B` a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the `test` argument is not a permutation test.
- `debug` a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
- `optimized` a boolean value. See `bnlearn-package` for details.
- `strict` a boolean value. If `TRUE` conflicting results in the learning process generate an error; otherwise they result in a warning.

Value

An object of class `bn`. See `bn-class` for details.

Author(s)

Marco Scutari

References

Tsamardinos I, Aliferis CF, Statnikov A (2003). "Time and Sample Efficient Discovery of Markov Blankets and Direct Causal Relations". In "KDD '03: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining", pp. 673-678. ACM.

Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm". *Machine Learning*, **65**(1), 31-78.

Aliferis FC, Statnikov A, Tsamardinos I, Subramani M, Koutsoukos XD (2010). "Local Causal and Markov Blanket Induction for Causal Discovery and Feature Selection for Classification Part I: Algorithms and Empirical Evaluation". *Journal of Machine Learning Research*, **11**, 171-234.

Margolin AA, Nemenman I, Basso K, Wiggins C, Stolovitzky G, Dalla Favera R, Califano A (2006). "ARACNE: An Algorithm for the Reconstruction of Gene Regulatory Networks in a Mammalian Cellular Context". *BMC Bioinformatics*, **7**(Suppl 1):S7.

See Also

constraint-based algorithms, score-based algorithms, hybrid algorithms.

`marks` *Examination marks data set*

Description

Examination marks of 88 students on five different topics, from Mardia (1979).

Usage

```
data(marks)
```

Format

The `marks` data set contains the following variables, one for each topic in the examination:

- `MECH` (*mechanics*)
- `VECT` (*vectors*)
- `ALG` (*algebra*)
- `ANL` (*analysis*)
- `STAT` (*statistics*)

All are measured on the same scale (0-100).

Source

Edwards DI (2000). *Introduction to Graphical Modelling*. Springer, 2nd edition.

Mardia KV, Kent JT, Bibby JM (1979). *Multivariate Analysis*. Academic Press.

Whittaker J (1990). *Graphical Models in Applied Multivariate Statistics*. Wiley.

Examples

```
# This is the undirected graphical model from Edwards (2000).
data(marks)
ug = empty.graph(names(marks))
arcs(ug, ignore.cycles = TRUE) = matrix(
  c("MECH", "VECT", "MECH", "ALG", "VECT", "MECH", "VECT", "ALG",
    "ALG",  "MECH", "ALG", "VECT", "ALG",  "ANL", "ALG",  "STAT",
    "ANL",  "ALG", "ANL",  "STAT", "STAT", "ALG", "STAT", "ANL"),
  ncol = 2, byrow = TRUE,
  dimnames = list(c(), c("from", "to")))
```

misc utilities *Miscellaneous utilities*

Description

Assign or extract various quantities of interest from an object of class bn of bn.fit.

Usage

```
## nodes
mb(x, node)
nbr(x, node)
parents(x, node)
parents(x, node, debug = FALSE) <- value
children(x, node)
children(x, node, debug = FALSE) <- value
in.degree(x, node)
out.degree(x, node)
# degree(x, node)
root.nodes(x)
leaf.nodes(x)

## arcs
arcs(x)
arcs(x, ignore.cycles = FALSE, debug = FALSE) <- value
directed.arcs(x)
undirected.arcs(x)
incoming.arcs(x, node)
outgoing.arcs(x, node)
incident.arcs(x, node)
narcs(x)

## adjacency matrix
amat(x)
amat(x, ignore.cycles = FALSE, debug = FALSE) <- value

## graphs
nparams(x, data, debug = FALSE)
ntests(x)
whitelist(x)
blacklist(x)

# shared with the graph package.
# these used to be a simple nodes(x) function.
## S4 method for signature 'bn'
nodes(object)
## S4 method for signature 'bn.fit'
```

```
nodes(object)
# these used to be a simple degree(x, node) function.
## S4 method for signature 'bn,ANY'
degree(object, Nodes)
## S4 method for signature 'bn.fit,ANY'
degree(object, Nodes)
```

Arguments

`x,object`	an object of class `bn` or `bn.fit`. The replacement form of `parents`, `children`, `arcs` and `amat` require an object of class `bn`.
`node,Nodes`	a character string, the label of a node.
`value`	either a vector of character strings (for `parents` and `children`), an adjacency matrix (for `amat`) or a data frame with two columns (optionally labeled "from" and "to", for `arcs`).
`data`	a data frame containing the data the Bayesian network was learned from. It's only needed if `x` is an object of class `bn`.
`ignore.cycles`	a boolean value. If `TRUE` the returned network will not be checked for cycles.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The number of parameters of a discrete Bayesian network is defined as the sum of the number of logically independent parameters of each node given its parents (Chickering, 1995). For Gaussian Bayesian networks the distribution of each node can be viewed as a linear regression, so it has a number of parameters equal to the number of the parents of the node plus one (the intercept) as per Neapolitan (2003).

Value

`mb`, `nbr`, `nodes`, `parents`, `children`, `root.nodes` and `leaf.nodes` return a vector of character strings.

`arcs`, `directed.arcs`, `undirected.arcs`, `incoming.arcs`, `outgoing.arcs`, `incident.arcs`, `whitelist` and `blacklist` return a matrix of two columns of character strings.

`narcs` returns the number of arcs in the graph.

`amat` returns a matrix of 0/1 integer values.

`degree`, `in.degree`, `out.degree`, `nparams` and `ntests` return an integer.

Author(s)

Marco Scutari

References

Chickering DM (1995). "A Transformational Characterization of Equivalent Bayesian Network Structures". In "UAI '95: Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence", pp. 87-98. Morgan Kaufmann.

Neapolitan RE (2003). *Learning Bayesian Networks*. Prentice Hall.

Examples

```
data(learning.test)
res = gs(learning.test)

##  the Markov blanket of A.
mb(res, "A")
# [1] "B" "D" "C"
## the neighbourhood of F.
nbr(res, "F")
# [1] "E"
## the arcs in the graph.
arcs(res)
#      from to
# [1,] "A"  "B"
# [2,] "A"  "D"
# [3,] "B"  "A"
# [4,] "B"  "E"
# [5,] "C"  "D"
# [6,] "F"  "E"
## the nodes of the graph.
nodes(res)
# [1] "A" "B" "C" "D" "E" "F"
## the adjacency matrix for the nodes of the graph.
amat(res)
#   A B C D E F
# A 0 1 0 1 0 0
# B 1 0 0 0 1 0
# C 0 0 0 1 0 0
# D 0 0 0 0 0 0
# E 0 0 0 0 0 0
# F 0 0 0 0 1 0
## the parents of D.
parents(res, "D")
# [1] "A" "C"
## the children of A.
children(res, "A")
# [1] "D"
## the root nodes of the graph.
root.nodes(res)
# [1] "C" "F"
## the leaf nodes of the graph.
leaf.nodes(res)
# [1] "D" "E"
## number of parameters of the Bayesian network.
```

```
res = set.arc(res, "A", "B")
nparams(res, learning.test)
# [1] 41
```

`model string utilities` *Build a model string from a Bayesian network and vice versa*

Description

Build a model string from a Bayesian network and vice versa.

Usage

```
modelstring(x)
modelstring(x, debug = FALSE) <- value

model2network(string, ordering = NULL, debug = FALSE)

## S3 method for class 'bn'
as.character(x, ...)
## S3 method for class 'character'
as.bn(x)
```

Arguments

`x`	an object of class `bn`. `modelstring` (but not its replacement form) accepts also objects of class `bn.fit`.
`string`	a character string describing the Bayesian network.
`ordering`	the labels of all the nodes in the graph; their order is the node ordering used in the construction of the `bn` object. If `NULL` the nodes are sorted alphabetically.
`value`	a character string, the same as the `string`.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
`...`	extra arguments from the generic method (currently ignored).

Details

The strings returned by `modelstring` have the same format as the ones returned by the `modelstring` function in package **deal**; network structures may be easily exported to and imported from that package (via the `model2network` function).

Value

`model2network` and `as.bn` return an object of class `bn`; `modelstring` and `as.character.bn` return a character string.

Author(s)

Marco Scutari

Examples

```
data(learning.test)
res = set.arc(gs(learning.test), "A", "B")
res
#
#   Bayesian network learned via Constraint-based methods
#
#   model:
#     [A][C][F][B|A][D|A:C][E|B:F]
#   nodes:                                 6
#   arcs:                                  5
#     undirected arcs:                     0
#     directed arcs:                       5
#   average markov blanket size:           2.33
#   average neighbourhood size:            1.67
#   average branching factor:              0.83
#
#   learning algorithm:                    Grow-Shrink
#   conditional independence test:         Mutual Information (discrete)
#   alpha threshold:                       0.05
#   tests used in the learning procedure:  43
#
modelstring(res)
# [1] "[A][C][F][B|A][D|A:C][E|B:F]"
res2 = model2network(modelstring(res))
res2
#
#   Random/Generated Bayesian network
#
#   model:
#     [A][C][F][B|A][D|A:C][E|B:F]
#   nodes:                                 6
#   arcs:                                  5
#     undirected arcs:                     0
#     directed arcs:                       5
#   average markov blanket size:           2.33
#   average neighbourhood size:            1.67
#   average branching factor:              0.83
#
#   generation algorithm:                  Empty
#
all.equal(res, res2)
# [1] TRUE
```

`naive.bayes` *Naive Bayes classifiers*

Description

Create, fit and perform predictions with naive Bayes and Tree-Augmented naive Bayes (TAN) classifiers.

Usage

```
naive.bayes(training, explanatory, data)
## S3 method for class 'bn.naive'
predict(object, data, prior, ..., prob = FALSE, debug = FALSE)

tree.bayes(x, training, explanatory, whitelist = NULL, blacklist = NULL,
  mi = NULL, root = NULL, debug = FALSE)
## S3 method for class 'bn.tan'
predict(object, data, prior, ..., prob = FALSE, debug = FALSE)
```

Arguments

`training`	a character string, the label of the training variable.
`explanatory`	a vector of character strings, the labels of the explanatory variables.
`object`	an object of class `bn.naive`, either fitted or not.
`x, data`	a data frame containing the variables in the model, which must all be factors.
`prior`	a numeric vector, the prior distribution for the training variable. It is automatically normalized if not already so.
`whitelist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`mi`	a character string, the estimator used for the mutual information coefficients for the Chow-Liu algorithm in TAN. Possible values are `mi` (discrete mutual information) and `mi-g` (Gaussian mutual information).
`root`	a character string, the label of the explanatory variable to bre used as the root of the tree in the TAN classifier.
`...`	extra arguments from the generic method (currently ignored).
`prob`	a boolean value. If `TRUE` the posterior probabilities used for prediction are attached to the predicted values as an attribute called `prob`.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The `naive.bayes` functions creates the star-shaped Bayesian network form of a naive Bayes classifier; the training variable (the one holding the group each observation belongs to) is at the center of the star, and it has an outgoing arc for each explanatory variable.

If `data` is specified, `explanatory` will be ignored and the labels of the explanatory variables will be extracted from the data.

predict performs a supervised classification of the observations by assigning them to the group with the maximum posterior probability.

Value

naive.bayes returns an object of class c("bn.naive", "bn"), which behaves like a normal bn object unless passed to predict. tree.bayes returns an object of class c("bn.tan", "bn"), which again behaves like a normal bn object unless passed to predict.

predict returns a factor with the same levels as the training variable from data. If prob = TRUE, the posterior probabilities used for prediction are attached to the predicted values as an attribute called prob.

Note

Since **bnlearn** does not support networks containing both continuous and discrete variables, all variables in data must be discrete.

Ties in prediction are broken using *Bayesian tie breaking*, i.e. sampling at random from the tied values. Therefore, setting the random seed is required to get reproducible results.

Author(s)

Marco Scutari

References

Borgelt C, Kruse R, Steinbrecher M (2009). *Graphical Models: Representations for Learning, Reasoning and Data Mining*. Wiley, 2nd edition.

Friedman N, Geiger D, Goldszmidt M (1997). "Bayesian Network Classifiers". *Machine Learning*, **29**(2–3), 131–163.

Examples

```
data(learning.test)
bn = naive.bayes("A", LETTERS[2:6])
pred = predict(bn, learning.test)
table(pred, learning.test[, "A"])
# pred    a    b    c
#    a 1286  310  178
#    b  192  977  203
#    c  190  383 1281

tan = tree.bayes(learning.test, "A")

fitted = bn.fit(tan, learning.test, method = "bayes")
pred = predict(fitted, learning.test)
table(pred, learning.test[, "A"])
# pred    a    b    c
#    a 1300  180  184
#    b  206 1342  239
#    c  162  148 1239
```

---

node ordering utilities

Utilities dealing with partial node orderings

Description

Find the partial node ordering implied by a network or generate the blacklist implied by a complete node ordering.

Usage

```
node.ordering(x, debug = FALSE)
ordering2blacklist(nodes)
tiers2blacklist(nodes)
```

Arguments

`x`	an object of class `bn` or `bn.fit`.
`nodes`	a node ordering, see below.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

`ordering2blacklist` takes a vector of character strings (the labels of the nodes), which specifies a complete node ordering. An object of class `bn` or `bn.fit`; in that case, the node ordering is derived by the graph. In both cases, the blacklist returned by `ordering2blacklist` contains all the possible arcs that violate the specified node ordering.

`tiers2blacklist` takes (again) a vector of character strings (the labels of the nodes), which specifies a complete node ordering, or a list of character vectors, which specifies a partial node ordering. In the latter case, all arcs going from a node in a particular element of the list (sometimes known as *tier*) to a node in one of the previous elements are blacklisted. Arcs between nodes in the same element are not blacklisted.

Value

`node.ordering` return a vector of character strings, an ordered set of node labels.

`ordering2blacklist` returns a sanitized `blacklist` (a two-column matrix, whose columns are labeled `from` and `to`).

Note

`node.ordering` and `ordering2blacklist` support only completely directed Bayesian networks.

Author(s)

Marco Scutari

Examples

```
data(learning.test)
res = gs(learning.test, optimized = TRUE)
res$learning$ntests
# [1] 43
res = set.arc(res, "A", "B")
ord = node.ordering(res)
ord
# [1] "A" "C" "F" "B" "D" "E"

## partial node ordering saves us two tests in the v-structure
## detection step of the algorithm.
gs(learning.test, blacklist = ordering2blacklist(ord))$learning$ntests
# [1] 41

tiers2blacklist(list(LETTERS[1:3], LETTERS[4:6]))
#       from to
#  [1,] "D"  "A"
#  [2,] "E"  "A"
#  [3,] "F"  "A"
#  [4,] "D"  "B"
#  [5,] "E"  "B"
#  [6,] "F"  "B"
#  [7,] "D"  "C"
#  [8,] "E"  "C"
#  [9,] "F"  "C"
```

---

plot.bn *Plot a Bayesian network*

Description

Plot the graph associated with a small Bayesian network.

Usage

```
## S3 method for class 'bn'
plot(x, ylim = c(0,600), xlim = ylim, radius = 250,
  arrow = 35, highlight = NULL, color = "red", ...)
```

Arguments

x	an object of class bn.
ylim	a numeric vector with two components containing the range on y-axis.
xlim	a numeric vector with two components containing the range on x-axis.
radius	a numeric value containing the radius of the nodes.
arrow	a numeric value containing the length of the arrow heads.

`highlight`	a vector of character strings, representing the labels of the nodes (and corresponding arcs) to be highlighted.
`color`	an integer or character string (the highlight colour).
`...`	other parameters to be passed through to plotting functions.

Note

The following graphical parameters are always overridden:

- `axes` is set to `FALSE`.
- `xlab` is set to an empty string.
- `ylab` is set to an empty string.

Author(s)

Marco Scutari

See Also

`graphviz.plot`.

Examples

```
data(learning.test)
res = gs(learning.test)

plot(res)

## highlight node B and related arcs.
plot(res, highlight = "B")
## highlight B and its Markov blanket.
plot(res, highlight = c("B", mb(res, "B")))

## a more compact plot.
par(oma = rep(0, 4), mar = rep(0, 4), mai = rep(0, 4),
  plt = c(0.06, 0.94, 0.12, 0.88))
plot(res)
```

`rbn` *Generate random data from a given Bayesian network*

Description

Generate random data from a given Bayesian network.

Usage

```
## S3 method for class 'bn'
rbn(x, n = 1, data, fit = "mle", ..., debug = FALSE)
## S3 method for class 'bn.fit'
rbn(x, n = 1, ..., debug = FALSE)
```

Arguments

`x`	an object of class `bn` or `bn.fit`.
`n`	a positive integer giving the number of observations to generate.
`data`	a data frame containing the data the Bayesian network was learned from.
`fit`	a character string, the label of the method used to fit the parameters of the newtork. See `bn.fit` for details.
`...`	additional arguments for the parameter estimation prcoedure, see again `bn.fit` for details..
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

A data frame with the same structure (column names and data types) of the `data` parameter.

Author(s)

Marco Scutari

References

Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.

See Also

`bn.boot`, `bn.cv`.

Examples

```
## Not run:
data(learning.test)
res = gs(learning.test)
res = set.arc(res, "A", "B")
par(mfrow = c(1,2))
plot(res)
sim = rbn(res, 500, learning.test)
plot(gs(sim))
## End(Not run)
```

score *Score of the Bayesian network*

Description

Compute the score of the Bayesian network.

Usage

```
score(x, data, type = NULL, ..., debug = FALSE)

## S3 method for class 'bn'
logLik(object, data, ...)
## S3 method for class 'bn'
AIC(object, data, ..., k = 1)
## S3 method for class 'bn'
BIC(object, data, ...)
```

Arguments

`x, object`	an object of class `bn`.
`data`	a data frame containing the data the Bayesian network was learned from.
`type`	a character string, the label of a network score. If none is specified, the default score is the *Bayesian Information Criterion* for both discrete and continuous data sets. See `bnlearn-package` for details.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
`...`	extra arguments from the generic method (for the `AIC` and `logLik` functions, currently ignored) or additional tuning parameters (for the `score` function).
`k`	a numeric value, the penalty per parameter to be used; the default `k = 1` gives the expression used to compute the AIC in the context of scoring Bayesian networks.

Details

Additional parameters of the `score` function:

- `iss`: the imaginary sample size, used by the Bayesian Dirichlet equivalent score (both the `bde` and `mbde`) and the Bayesian Gaussian posterior density. It is also known as "equivalent sample size". The default value is equal to `10` for both the `bde`/`mbde` scores and `bge`.
- `exp`: a list of indexes of experimental observations (those that have been artificially manipulated). Each element of the list must be named after one of the nodes, and must contain a numeric vector with indexes of the observations whose value has been manipulated for that node.
- `k`: the penalty per parameter to be used by the AIC and BIC scores. The default value is `1` for AIC and `log(nrow(data))/2` for BIC.

- phi: the prior phi matrix formula to use in the Bayesian Gaussian equivalent (bge) score. Possible values are heckerman (default) and bottcher (the one used by default in the **deal** package.)

Value

A numeric value, the score of the Bayesian network.

Author(s)

Marco Scutari

References

Chickering DM (1995). "A Transformational Characterization of Equivalent Bayesian Network Structures". In "UAI '95: Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence", pp. 87-98. Morgan Kaufmann.

Geiger D, Heckerman D (1994). "Learning Gaussian Networks". *Technical report*, Microsoft Research. Available as Technical Report MSR-TR-94-10.

Heckerman D, Geiger D, Chickering DM (1995). "Learning Bayesian Networks: The Combination of Knowledge and Statistical Data". *Machine Learning*, **20**(3), 197-243. Available as Technical Report MSR-TR-94-09.

Cooper GF, Yoo C (1999). "Causal Discovery from a Mixture of Experimental and Observational Data". In "UAI '99: Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence", pp. 116-125. Morgann Kaufmann.

See Also

choose.direction, arc.strength.

Examples

```
data(learning.test)
res = set.arc(gs(learning.test), "A", "B")
score(res, learning.test, type = "bde")
# [1] -23967.65
## let's see score equivalence in action!
res2 = set.arc(gs(learning.test), "B", "A")
score(res2, learning.test, type = "bde")
# [1] -23967.65

## k2 score on the other hand is not score equivalent.
score(res, learning.test, type = "k2")
# [1] -23958.70
score(res2, learning.test, type = "k2")
# [1] -23957.68

## equivalent to logLik(res, learning.test)
score(res, learning.test, type = "loglik")
# [1] -23832.13
```

```
## equivalent to AIC(res, learning.test)
score(res, learning.test, type = "aic")
# [1] -23873.13
```

---

score-based algorithms

Score-based structure learning algorithms

Description

Learn the structure of a Bayesian network using a hill-climbing (HC) or a Tabu search (TABU) greedy search.

Usage

```
hc(x, start = NULL, whitelist = NULL, blacklist = NULL,
  score = NULL, ..., debug = FALSE, restart = 0,
  perturb = 1, max.iter = Inf, optimized = TRUE)
tabu(x, start = NULL, whitelist = NULL, blacklist = NULL,
  score = NULL, ..., debug = FALSE, tabu = 10, max.tabu = tabu,
  max.iter = Inf, optimized = TRUE)
```

Arguments

`x`	a data frame containing the variables in the model.
`start`	an object of class bn, the preseeded directed acyclic graph used to initialize the algorithm. If none is specified, an empty one (i.e. without any arc) is used.
`whitelist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`score`	a character string, the label of the network score to be used in the algorithm. If none is specified, the default score is the*Bayesian Information Criterion* for both discrete and continuous data sets. See `bnlearn-package` for details.
`...`	additional tuning parameters for the network score. See `score` for details.
`debug`	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
`restart`	an integer, the number of random restarts.
`tabu`	a positive integer number, the length of the tabu list used in the `tabu` function.
`max.tabu`	a positive integer number, the iterations tabu search can perform without improving the best network score.
`perturb`	an integer, the number of attempts to randomly insert/remove/reverse an arc on every random restart.
`max.iter`	an integer, the maximum number of iterations.
`optimized`	a boolean value. See `bnlearn-package` for details.

Value

An object of class bn. See bn-class for details.

Author(s)

Marco Scutari

References

Russell SJ, Norvig P (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.

Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.

Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-03-153.

Daly R, Shen Q (2007). "Methods to Accelerate the Learning of Bayesian Network Structures". In "Proceedings of the 2007 UK Workshop on Computational Intelligence", Imperial College, London.

See Also

constraint-based algorithms, hybrid algorithms,
local discovery algorithms.

---

single-node local discovery

Discover the structure around a single node

Description

Learn the Markov blanket or the neighbourhood centered on a node.

Usage

```
learn.mb(x, node, method, whitelist = NULL, blacklist = NULL, start = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE, optimized = TRUE)
learn.mb(x, node, method, whitelist = NULL, blacklist = NULL, start = NULL,
  test = NULL, alpha = 0.05, B = NULL, debug = FALSE, optimized = TRUE)
```

Arguments

x	a data frame containing the variables in the model.
node	a character string, the label of the node whose local structure is being learned.
method	a character string, the label of a structure learning algorithm. Possible choices are constraint-based algorithms for learn.mb and local discovery algorithms for learn.nbr.
whitelist	a vector of character strings, the labels of the whitelisted nodes.

blacklist	a vector of character strings, the labels of the blacklisted nodes.
start	a vector of character strings, the labels of the nodes to be included in the Markov blanket before the learning process (in `learn.mb`). Note that the nodes in `start` can be removed from the Markov blanket by the learning algorithm, unlike the nodes included due to whitelisting.
test	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for discrete data sets and the *linear correlation* for continuous ones. See `bnlearn-package` for details.
alpha	a numeric value, the target nominal type I error rate.
B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the `test` argument is not a permutation test.
debug	a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.
optimized	a boolean value. See `bnlearn-package` for details.

Value

A vector of character strings, the labels of the nodes in the Markov blanket (for `learn.mb`) or in the neighbourhood (for `learn.nbr`).

Author(s)

Marco Scutari

See Also

constraint-based algorithms, local discovery algorithms.

`snow integration` *bnlearn - snow package integration*

Description

How to use the **bnlearn** package with the parallel computing environment provided by the **snow** package.

Parallel computing for constraint-based algorithms

```
# load snow, bnlearn and rsprng (for parallel random number
# generation, just in case it's needed); start LAM/MPI via
# lamboot if using an MPI cluster.
> library(snow)
> library(bnlearn)
> library(rsprng)
```

```
# initialize the cluster ("socket" and "PVM" clusters are fine, too).
> cl <- makeCluster(2, type = "MPI")
Loading required package: Rmpi
        2 slaves are spawned successfully. 0 failed.
> clusterSetupSPRNG(cl)
# load the data.
> data(learning.test)
# call a learning function passing the cluster object (the
# return value of the previous makeCluster() call) as a
# parameter.
> res = gs(learning.test, cluster = cl)
# note that the number of test is evenly divided between
# the two nodes of the cluster.
> clusterEvalQ(cl, .test.counter)
[[1]]
[1] 47

[[2]]
[1] 42
# a few tests are still executed by this process.
> .test.counter
[1] 4
# stop the cluster.
> stopCluster(cl)
[1] 1
```

Author(s)

Marco Scutari

---

strength.plot *Arc strength plot*

Description

Plot a Bayesian network and format its arcs according to the strength of the dependencies they represent. Requires the **Rgraphviz** package.

Usage

```
strength.plot(x, strength, threshold, cutpoints, highlight = NULL,
  layout = "dot", shape = "circle", main = NULL, sub = NULL,
  debug = FALSE)
```

Arguments

- `x` — an object of class `bn`.
- `strength` — an object of class `bn.strength` computed from the object of class `bn` corresponding to the `x` parameter.
- `threshold` — a numeric value. See below.
- `cutpoints` — an array of numeric values. See below.
- `highlight` — a list, see `graphviz.plot` for details.
- `layout` — a character string, the layout parameter to be passed to **Rgraphviz**. Possible values are `dots`, `neato`, `twopi`, `circo` and `fdp`. See **Rgraphviz** documentation for details.
- `shape` — a character string, the shape of the nodes. Can be either `circle` or `ellipse`.
- `main` — a character string, the main title of the graph. It's plotted at the top of the graph.
- `sub` — a character string, a subtitle which is plotted at the bottom of the graph.
- `debug` — a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Details

The `threshold` parameter is used to determine which arcs are supported strongly enough by the data to be deemed significant:

- if arc strengths have been computed using conditional independence tests, any strength coefficient (which is the p-value of the test) lesser or equal than the threshold is considered significant. In this case the default value of `threshold` is equal to the value of the `alpha` parameter used in the call to `arc.strength`, which in turn defaults to the one used by the learning algorithm (if any) or to `0.05`.
- if arc strengths have been computed using network scores, any strength coefficient (which is the increase/decrease of the network score caused by the removal of the arc) lesser than the threshold is considered significant. In this case the default value of `threshold` is `0`.
- if arc strengths have been computed using bootstrap, any strength coefficient (which is the relative frequency of the arc in the networks learned from the bootstrap replicates) greater or equal than the threshold is considered significant. In this case the default value of `threshold` is `0.5`.

Non-significant arcs are plotted as dashed lines.

The `cutpoints` parameter is an array of numeric values used to divide the range of the strength coefficients into intervals. The interval each strength coefficient falls into determines the line width of the corresponding arc in the plot. The default intervals are delimited by

```
unique(c(0, threshold/c(10, 5, 2, 1.5, 1), 1))
```

if the coefficients are computed from conditional independence tests, by

```
1 - unique(c(0, threshold/c(10, 5, 2, 1.5, 1), 1))
```

for bootstrap estimates or by the quantiles

```
quantile(-s[s < threshold], c(0.50, 0.75, 0.90, 0.95, 1))
```

of the significant coefficients if network scores are used.

Value

The object of class graphAM used to format and render the plot. It can be further modified using the commands present in the **graph** and **Rgraphviz** packages.

Author(s)

Marco Scutari

Examples

```
## Not run:
# plot the network learned by gs().
res = set.arc(gs(learning.test), "A", "B")
strength = arc.strength(res, learning.test, criterion = "x2")
strength.plot(res, strength)
# add another (non-significant) arc and plot the network again.
res = set.arc(res, "A", "C")
strength = arc.strength(res, learning.test, criterion = "x2")
strength.plot(res, strength)

## End(Not run)
```

Index