

Package ‘bnlearn’

June 14, 2010

Type Package

Title Bayesian network structure learning

Version 2.1

Date 2010-06-13

Depends R (>= 2.8.0), utils,

Suggests snow, graph, Rgraphviz (>= 1.20.3), lattice

Author Marco Scutari

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

Description Bayesian network structure learning via constraint-based (also known as ‘conditional independence’), score-based and hybrid algorithms. 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 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. Some utility functions (model comparison and manipulation, random data generation, arc orientation testing, simple and advanced plots) are included, as well as support for basic parametric and bootstrap inference, conditional probability queries and cross-validation.

URL <http://www.bnlearn.com/>

License GPL (>= 2)

LazyLoad yes

LazyData yes

Repository CRAN

Date/Publication 2010-06-14 06:55:27

R topics documented:

bnlearn-package	3
alarm	9
arc operations	11
arc.strength	12
asia	14
bn class	15
bn.boot	17
bn.cv	18
bn.fit	20
bn.fit class	21
bn.fit plots	22
bn.fit utilities	23
bn.kcv class	26
bn.strength class	26
bn.var	27
boot.strength	29
choose.direction	30
ci.test	31
compare	34
constraint-based algorithms	35
coronary	37
cpdag	38
cpquery	39
deal integration	40
gaussian.test	41
graph generation utilities	42
graph utilities	44
graphviz.plot	46
hailfinder	47
hybrid algorithms	51
insurance	53
learning.test	55
lizards	56
local discovery algorithms	57
marks	59
misc utilities	60
model string utilities	62
node ordering utilities	64
plot.bn	65
rbn	66
score	68
score-based algorithms	70
snow integration	71
strength.plot	72

bnlearn-package *Bayesian network structure learning.*

Description

Bayesian network structure learning via constraint-based, score-based and hybrid algorithms.

Details

Package: bnlearn
Type: Package
Version: 2.1
Date: 2010-06-13
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* (`rsmx2`): 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 *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).

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

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. Both the asymptotic χ^2 test (`mi`) and the Monte Carlo permutation test (`mc-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. Both the asymptotic χ^2 test (`x2`) and the Monte Carlo permutation test (`mc-x2`) are implemented.
 - *Akaike Information Criterion* (`aict`): an experimental AIC-based independence test, computed comparing the mutual information and the expected information gain.
- *continuous case* (multivariate normal distribution)
 - *linear correlation*: linear correlation. Both the exact Student's t test (`cor`) and the Monte Carlo permutation test (`mc-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). Both the asymptotic normal (`zf`) and the Monte Carlo permutation test (`mc-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). Both the asymptotic χ^2 test (`mi-g`) and the Monte Carlo permutation test (`mc-mi-g`) are implemented.

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 *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. $A \rightarrow B$ is whitelisted but $B \rightarrow A$ is not) have the respective reverse arcs blacklisted, and are always present in the graph.
- arcs whitelisted in both directions (i.e. both $A \rightarrow B$ and $B \rightarrow A$ 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.

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 procedure. This approach may hide errors either in the Markov blanket or the neighbourhood detection phase in some particular cases, 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 the Markov blanket and neighbour detection 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, which may help to retrieve a good model even 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).

Author(s)

Marco Scutari
Department of Statistical Sciences
University of Padova

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)

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

Korb K, Nicholson AE (2003). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC.

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.

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?
compare(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/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

<code>x</code>	an object of class <code>bn</code> .
<code>from</code>	a character string, the label of a node.
<code>to</code>	a character string, the label of another node.
<code>check.cycles</code>	a boolean value. If <code>TRUE</code> the graph is tested for acyclicity; otherwise the graph is returned anyway.
<code>debug</code>	a boolean value. If <code>TRUE</code> 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	<i>Measure the strength of the arcs present in the network</i>
--------------	--

Description

Strength of the probabilistic relations expressed by the arcs of the Bayesian network.

Usage

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

Arguments

<code>x</code>	an object of class <code>bn</code> .
<code>data</code>	a data frame containing the data the Bayesian network was learned from.
<code>criterion</code>	a character string, the label of a score function, the label of an independence test or <code>bootstrap</code> . See bnlearn-package for details on the first two possibilities.
<code>...</code>	additional tuning parameters for the network score (if <code>criterion</code> is the label of a score function, see score for details), the conditional independence test (currently the only one is <code>B</code> , the number of permutations) or the bootstrap simulation (if <code>criterion</code> is set to <code>bootstrap</code> , 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.

Value

`arc.strength` returns an object of class `bn.strength`. See [bn.strength class](#) for details.

Author(s)

Marco Scutari

See Also

[choose.direction](#), [score](#).

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
arc.strength(res, learning.test, criterion = "aic")
#   from to      strength
# 1   A  B -1166.9139
# 2   A  D -1978.0531
# 3   B  E  -746.8954
# 4   C  D  -862.8637
# 5   F  E  -568.7816
res = set.arc(res, "B", "A")
# A -> B and B -> A have the same strength because they
# are score equivalent.
arc.strength(res, learning.test, criterion = "aic")
```

```

#   from to   strength
# 1    A  D -1978.0531
# 2    B  E  -746.8954
# 3    C  D  -862.8637
# 4    F  E  -568.7816
# 5    B  A -1166.9139
## Not run:
arc.strength(res, data = learning.test, criterion = "bootstrap",
  R = 200, algorithm.args = list(alpha = 0.10))
#   from to strength
# 1    A  B         1
# 2    A  D         1
# 3    B  E         1
# 4    C  D         1
# 5    F  E         1

## 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 behave 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.

- `n_tests`: 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.
- `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 Bayesian networks 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(), debug = FALSE)
```

Arguments

<code>data</code>	a data frame containing the variables in the model.
<code>statistic</code>	a function or a character string (the name of a function) to be applied to each bootstrap replicate.
<code>R</code>	a positive integer, the number of bootstrap replicates.
<code>m</code>	a positive integer, the size of each bootstrap replicate.
<code>sim</code>	a character string indicating the type of simulation required. Possible values are "ordinary" (the default) and "parametric".
<code>algorithm</code>	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are <code>gs</code> , <code>iamb</code> , <code>fast.iamb</code> , <code>inter.iamb</code> , <code>mmpc</code> and <code>hc</code> . See bnlearn-package and documentation of each algorithm for details.
<code>algorithm.args</code>	a list of extra arguments to be passed to the learning algorithm.
<code>statistic.args</code>	a list of extra arguments to be passed to the function specified by <code>statistic</code> .
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

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(), 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 <code>bn</code> (a fixed network structure).
loss	a character string, the label of a loss function. If none is specified, the default loss function is the <i>Log-Likelihood Loss</i> 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.

<code>algorithm.args</code>	a list of extra arguments to be passed to the learning algorithm.
<code>loss.args</code>	a list of extra arguments to be passed to the loss function specified by <code>loss</code> .
<code>fit</code>	a character string, the label of the method used to fit the parameters of the network. See bn.fit for details.
<code>fit.args</code>	additional arguments for the parameter estimation procedure, see again bn.fit for details..
<code>debug</code>	a boolean value. If <code>TRUE</code> 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)
```

Arguments

x	an object of class <code>bn</code> .
data	a data frame containing the variables in the model.
method	a character string, either <code>mle</code> for <i>Maximum Likelihood parameter estimation</i> or <code>bayes</code> for <i>Bayesian parameter estimation</i> (currently implemented only for discrete data).
...	additional arguments for the parameter estimation procedure, see below.
debug	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

Value

An object of class `bn.fit`. See `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)
```

<code>bn.fit class</code>	<i>The bn.fit class structure</i>
---------------------------	-----------------------------------

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.

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

<code>fitted</code>	an object of class <code>bn.fit</code> , <code>bn.fit.dnode</code> or <code>bn.fit.gnode</code> .
<code>xlab</code> , <code>ylab</code> , <code>main</code>	the label of the x axis, of the y axis, and the plot title.
<code>density</code>	a boolean value. If <code>TRUE</code> the histogram is plotted using relative frequencies, and the matching normal density is added to the plot.
<code>...</code>	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, ...)
## S3 method for class 'bn.fit':
logLik(object, data, ...)
## S3 method for class 'bn.fit':
AIC(object, data, ..., k = 1)

## 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, ...)

## 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, ...)
```

Arguments

object	an object of class <code>bn.fit</code> , <code>bn.fit.dnode</code> or <code>bn.fit.gnode</code> .
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 <code>k = 1</code> gives the expression used to compute AIC.

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`).

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.

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](#) and [strength.plot](#) for details.

and some additional attributes:

`reduce` a character string, either `first` or `second`. If `first` all the arcs with first moment equal to zero are dropped; 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`).

`bn.var.test` returns an object of class `htest`.

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
bn.var.test(z, method = "nvar")
#
# Squared Frobenius Norm
#
# data: covariance matrix
# nvar = 0.5471, B = 5000, R = 100, p-value < 2.2e-16
# alternative hypothesis: true value is greater than 0

## End(Not run)
```

boot.strength	<i>Bootstrap arc strength and direction</i>
---------------	---

Description

Use nonparametric bootstrap to assess arc strength and direction.

Usage

```
boot.strength(data, R = 200, m = nrow(data),
              algorithm, algorithm.args = list(), debug = FALSE)
```

Arguments

data	a data frame containing the variables in the model.
R	a positive integer, the number of bootstrap replicates.
m	a positive integer, the size of each bootstrap replicate.
algorithm	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are <code>gs</code> , <code>iamb</code> , <code>fast.iamb</code> , <code>inter.iamb</code> , <code>mmpc</code> and <code>hc</code> . 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.
debug	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

Value

A 4-column data frame (very similar to an object of class `bn.strength`) with an entry for each possible arc in the network and the following columns:

from, to	the nodes incident on the arc.
strength	the strength of the arc, computed as the probability of observing an arc between <code>from</code> and <code>to</code> in the bootstrap replicates, regardless of its direction.
direction	the confidence in the direction of the arc, computed as the probability of that particular direction in the bootstrap replicates conditional on the presence of an arc between <code>from</code> and <code>to</code> (again regardless of its direction).

Author(s)

Marco Scutari

References

Imoto S, Kim SY, Shimodaira H, Aburatani S, Tashiro K, Kuhara S, Miyano S (2002). "Bootstrap Analysis of Gene Networks Based on Bayesian Networks and Nonparametric Regression". *Genome Informatics*, **13**, 369-370.

See Also

[arc.strength](#), [bn.boot](#).

`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

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

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.

```

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 <i>mutual information</i> for discrete data sets and the <i>linear correlation</i> 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 <code>test</code> 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, NA if the p-value is computed by Monte Carlo simulation.
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.

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

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.

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 or compute the Structural Hamming Distance (SHD) between them.

Usage

```
compare(r1, r2, debug = FALSE)
shd(learned, true, debug = FALSE)
```

Arguments

`r1`, `learned` an object of class `bn`.
`r2`, `true` another object of class `bn`.
`debug` a boolean value. If `TRUE` a lot of debugging output is printed; otherwise the function is completely silent.

Value

`compare` returns a boolean value (`TRUE` if the objects describe the same network structure, `FALSE` otherwise). `shd` returns 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)

res = gs(learning.test)
## the arc between A and B has no direction
plot(res, highlight = c("A", "B"))
res2 = set.arc(res, "A", "B")
compare(res, res2, debug = TRUE)
# * children of A in r1 not present in r2:
# character(0)
# * children of A in r2 not present in r1:
# [1] "B"
# * parents of B in r1 not present in r2:
```

```

# character(0)
# * parents of B in r2 not present in r1:
# [1] "A"
# * directed arcs in r1 not present in r2:
# character(0)
# * directed arcs in r2 not present in r1:
# [1] "A -> B"
# * undirected arcs in r1 not present in r2:
# [1] "A - B" "B - A"
# * undirected arcs in r2 not present in r1:
# character(0)
# [1] FALSE
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.
# * arcs between E and F do not match.
# [1] 2

```

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.

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.

<code>cluster</code>	an optional cluster object from package <code>snow</code> . See snow integration for details and a simple example.
<code>whitelist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
<code>blacklist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
<code>test</code>	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 <i>mutual information</i> for discrete data sets and the <i>linear correlation</i> for continuous ones. See bnlearn-package for details.
<code>alpha</code>	a numeric value, the target nominal type I error rate.
<code>B</code>	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 <code>test</code> argument is not a permutation test.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.
<code>optimized</code>	a boolean value. See bnlearn-package for details.
<code>strict</code>	a boolean value. If <code>TRUE</code> conflicting results in the learning process generate an error; otherwise they result in a warning.
<code>undirected</code>	a boolean value. If <code>TRUE</code> 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 thrombosis, 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

Find the equivalence class of a Bayesian network

Description

Find the equivalence class and the v-structures of a Bayesian network, or construct its moral graph.

Usage

```

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

```

Arguments

x	an object of class <code>bn</code> .
arcs	a boolean value. If <code>TRUE</code> the arcs that are part of at least one v-structure are returned instead of the v-structures themselves.
debug	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

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.

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

Author(s)

Marco Scutari

References

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, method = "ls", ..., debug = FALSE)
```

Arguments

fitted	an object of class <code>bn.fit</code> .
event, evidence	see below.
method	a character string, the method used to perform the conditional probability query. Currently only <i>Logic Sampling</i> is implemented.
...	additional tuning parameters.
debug	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

Value

A numeric value, the conditional probability of event 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`.

Author(s)

Marco Scutari

References

Korb K, Nicholson AE (2003). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC.

Examples

```
fitted = bn.fit(hc(learning.test), learning.test)
cpquery(fitted, (B == "b"), (A == "a"))
# the result should be around 0.025.
```

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

gaussian.test

Synthetic (continuous) data set to test learning algorithms

Description

This is 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)
```

graph generation utilities

Generate empty or random graphs

Description

Generate empty or random 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 <i>ordered</i> (<i>full ordering</i> based generation), <i>ic-dag</i> (Ide’s and Cozman’s <i>Generating Multi-connected DAGs</i> algorithm), <i>melancon</i> (Melancon’s and Philippe’s <i>Uniform Random Acyclic Digraphs</i> algorithm) and <i>empty</i> (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.
- *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 $2 / (\text{length}(\text{nodes}) - 1)$, 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 $6 * \text{length}(\text{nodes})^2$.
- `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).

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 (2000). "Random Generation of Dags for Graph Drawing". Technical Report INS-R0005, Centre for Mathematics and Computer Sciences, Amsterdam.

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])
#
# Randomly 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 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, directed, 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)

```

Arguments

<code>x</code>	an object of class <code>bn</code> . <code>acyclic</code> , <code>directed</code> and <code>path</code> also accept objects of class <code>bn.fit</code> .
<code>from</code>	a character string, the label of a node.
<code>to</code>	a character string, the label of a node (different from <code>from</code>).
<code>direct</code>	a boolean value. If <code>FALSE</code> ignore any arc between <code>from</code> and <code>to</code> when looking for a path.
<code>directed</code>	a boolean value. If <code>TRUE</code> the graph is assumed to be completely directed (no undirected arcs), and a faster cycle detection algorithm is used.
<code>underlying.graph</code>	a boolean value. If <code>TRUE</code> the underlying undirected graph is used instead of the (directed) one from the <code>x</code> parameter.
<code>ordering</code>	the labels of all the nodes in the graph; their order is the node ordering used to set the direction of undirected arcs.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

Value

`acyclic`, `path` and `directed` return a boolean value.
`skeleton` and `pdag2dag` 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 <code>bn</code> .
highlight	a list, see below.
layout	a character string, the layout parameter to be passed to Rgraphviz . Possible values are <code>dots</code> , <code>neato</code> , <code>twopi</code> , <code>circo</code> and <code>fdp</code> . See Rgraphviz documentation for details.
shape	a character string, the shape of the nodes. Can be either <code>circle</code> or <code>ellipse</code> .
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). The default value is `red`.
- `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**.

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, Part Inhibit, Stifling and Total Inhibit.
- AMCINInScen (*AM convective inhibition in scenario*): a three-level factor with levels LessThanAve, Average and MoreThanAve.

- CapInScen (*capping within 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, NIgtLIgt_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.
- InsScIInScen (*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 (*relative 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/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

```

rsmx2(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

<code>x</code>	a data frame containing the variables in the model.
<code>whitelist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
<code>blacklist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
<code>restrict</code>	a character string, the constraint-based algorithm to be used in the “restrict” phase. Possible values are <code>gs</code> , <code>iamb</code> , <code>fast.iamb</code> , <code>inter.iamb</code> and <code>mmhc</code> . See bnlearn-package and the documentation of each algorithm for details.
<code>maximize</code>	a character string, the score-based algorithm to be used in the “maximize” phase. The only possible value is <code>hc</code> . See bnlearn-package for details.
<code>test</code>	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 <i>mutual information</i> for discrete data sets and the <i>linear correlation</i> for continuous ones. See bnlearn-package for details.
<code>score</code>	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 <i>Bayesian Information Criterion</i> for both discrete and continuous data sets. See bnlearn-package for details.

<code>alpha</code>	a numeric value, the target nominal type I error rate of the conditional independence test.
<code>B</code>	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 <code>test</code> argument is not a permutation test.
<code>...</code>	additional tuning parameters for the network score used by the score-based algorithm. See score for details.
<code>maximize.args</code>	a list of arguments to be passed to the score-based algorithm specified by <code>maximize</code> , such as <code>restart</code> for hill-climbing or <code>tabu</code> for tabu search.
<code>restart</code>	an integer, the number of random restarts for the score-based algorithm.
<code>perturb</code>	an integer, the number of attempts to randomly insert/remove/reverse an arc on every random restart.
<code>max.iter</code>	an integer, the maximum number of iterations for the score-based algorithm.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.
<code>optimized</code>	a boolean value. See bnlearn-package for details.
<code>strict</code>	a boolean value. If <code>TRUE</code> 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/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 underlying structure of a directed acyclic graph (DAG) from data using the Max-Min Parents and Children (MMPC) constraint-based algorithm.

Usage

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

<code>whitelist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
<code>blacklist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
<code>test</code>	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 <i>mutual information</i> for discrete data sets and the <i>linear correlation</i> for continuous ones. See bnlearn-package for details.
<code>alpha</code>	a numeric value, the target nominal type I error rate.
<code>B</code>	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 <code>test</code> argument is not a permutation test.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.
<code>optimized</code>	a boolean value. See bnlearn-package for details.
<code>strict</code>	a boolean value. If <code>TRUE</code> 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.

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")))
```

Description

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

Usage

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

## arcs
arcs(x)
arcs(x, ignore.cycles = FALSE, debug = FALSE) <- value
directed.arcs(x)
undirected.arcs(x)

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

## graphs
nparams(x, data, debug = FALSE)
```

Arguments

<code>x</code>	an object of class <code>bn</code> or <code>bn.fit</code> . The replacement form of <code>parents</code> , <code>children</code> , <code>arcs</code> and <code>amat</code> require an object of class <code>bn</code> .
<code>node</code>	a character string, the label of a node.
<code>value</code>	either a vector of character strings (for <code>parents</code> and <code>children</code>), an adjacency matrix (for <code>amat</code>) or a data frame with two columns (optionally labeled "from" and "to", for <code>arcs</code>).
<code>data</code>	a data frame containing the data the Bayesian network was learned from. It's only needed if <code>x</code> is an object of class <code>bn</code> .
<code>ignore.cycles</code>	a boolean value. If <code>TRUE</code> the returned network will not be checked for cycles.
<code>debug</code>	a boolean value. If <code>TRUE</code> 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`, `root.nodes` and `leaf.nodes` return a vector of character strings.
`arcs` returns a matrix of two columns of character strings.
`amat` returns a matrix of 0/1 integer values.
`nparams` returns 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, debug = FALSE)

## S3 method for class 'bn':
as.character(x, ...)
## S3 method for class 'character':
as.bn(string, debug = FALSE)

```

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.
value	a character string, the same as the <code>string</code> .
debug	a boolean value. If <code>TRUE</code> 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
#
# Randomly 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
#
compare(res, res2)
# [1] TRUE

```

node ordering utilities

Utilities dealing with partial node orderings

Description

Detect 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)

```

Arguments

x	an object of class <code>bn</code> . <code>node.ordering</code> also accepts objects of class <code>bn.fit</code> .
nodes	a vector of character strings, the labels of the nodes. The ordering of the labels must reflect the partial node ordering of the nodes in the graph.
debug	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.

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
```

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

<code>x</code>	an object of class <code>bn</code> or <code>bn.fit</code> .
<code>n</code>	a positive integer giving the number of observations to generate.
<code>data</code>	a data frame containing the data the Bayesian network was learned from.
<code>fit</code>	a character string, the label of the method used to fit the parameters of the network. See <code>bn.fit</code> for details.
<code>...</code>	additional arguments for the parameter estimation procedure, see again <code>bn.fit</code> for details..
<code>debug</code>	a boolean value. If <code>TRUE</code> 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 (2003). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC.

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)
```

Arguments

<code>x</code> , <code>object</code>	an object of class <code>bn</code> .
<code>data</code>	a data frame containing the data the Bayesian network was learned from.
<code>type</code>	a character string, the label of a network score. If none is specified, the default score is the <i>Akaike Information Criterion</i> for discrete data sets and the <i>Bayesian Information Criterion</i> for both discrete and continuous data sets. See bnlearn-package for details.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.
<code>...</code>	extra arguments from the generic method (for the <code>AIC</code> and <code>logLik</code> functions, currently ignored) or additional tuning parameters (for the <code>score</code> function).
<code>k</code>	a numeric value, the penalty per parameter to be used; the default <code>k = 1</code> 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 and the Bayesian Gaussian posterior density. It is also known as “equivalent sample size”. The default value is equal to the number of cells of the joint contingency table (for compatibility with the **deal** package) for the `bde` score, or to the number of independent parameters for the `bge` score.
- `k`: the penalty per parameter to be used by the AIC and BIC scores. The default value is 1 for AIC and $\log(\text{nrow}(\text{data}))/2$ 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.

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] -24002.36
## let's see score equivalence in action!
res2 = set.arc(gs(learning.test), "B", "A")
score(res2, learning.test, type = "bde")
# [1] -24002.36

## 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

<code>x</code>	a data frame containing the variables in the model.
<code>start</code>	an object of class <code>bn</code> , the preseeded directed acyclic graph used to initialize the algorithm. If none is specified, an empty one (i.e. without any arc) is used.
<code>whitelist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
<code>blacklist</code>	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
<code>score</code>	a character string, the label of the network score to be used in the algorithm. If none is specified, the default score is the <i>Bayesian Information Criterion</i> for both discrete and continuous data sets. See bnlearn-package for details.
<code>...</code>	additional tuning parameters for the network score. See score for details.
<code>debug</code>	a boolean value. If <code>TRUE</code> a lot of debugging output is printed; otherwise the function is completely silent.
<code>restart</code>	an integer, the number of random restarts.
<code>tabu</code>	a positive integer number, the length of the tabu list used in the <code>tabu</code> function.
<code>max.tabu</code>	a positive integer number, the iterations tabu search can perform without improving the best network score.
<code>perturb</code>	an integer, the number of attempts to randomly insert/remove/reverse an arc on every random restart.
<code>max.iter</code>	an integer, the maximum number of iterations.
<code>optimized</code>	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 (2003). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC.

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](#).

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] 46

[[2]]
[1] 40
# 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 <code>circle</code> or <code>ellipse</code> .
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 <code>TRUE</code> 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

*Topic **classes**

- bn class, 14
- bn.fit class, 20
- bn.kcv class, 25
- bn.strength class, 25

*Topic **datasets**

- alarm, 8
- asia, 13
- coronary, 36
- gaussian.test, 40
- hailfinder, 46
- insurance, 52
- learning.test, 54
- lizards, 55
- marks, 58

*Topic **documentation**

- deal integration, 39
- snow integration, 70

*Topic **graphs**

- arc operations, 10
- bn.fit utilities, 22
- compare, 33
- constraint-based algorithms, 34
- cpdag, 37
- graph generation utilities, 41
- graph utilities, 43
- hybrid algorithms, 50
- local discovery algorithms, 56
- misc utilities, 59
- model string utilities, 61
- score-based algorithms, 69

*Topic **hplot**

- bn.fit plots, 21
- graphviz.plot, 45
- plot.bn, 64
- strength.plot, 71

*Topic **htest**

- arc.strength, 11
- choose.direction, 29
- ci.test, 30
- score, 67

*Topic **models**

- constraint-based algorithms, 34
- hybrid algorithms, 50
- local discovery algorithms, 56
- score-based algorithms, 69

*Topic **multivariate**

- bn.boot, 16
- bn.cv, 17
- bn.fit, 19
- bn.var, 26
- boot.strength, 28
- constraint-based algorithms, 34
- cpdag, 37
- cpquery, 38
- hybrid algorithms, 50
- local discovery algorithms, 56
- node ordering utilities, 63
- rbn, 65
- score-based algorithms, 69

*Topic **nonparametric**

- bn.boot, 16
- bn.cv, 17
- boot.strength, 28

*Topic **package**

- bnlearn-package, 2

*Topic **utilities**

- arc operations, 10
- bn.fit utilities, 22
- graph generation utilities, 41

- graph utilities, 43
 - misc utilities, 59
 - model string utilities, 61
 - node ordering utilities, 63
 - rbn, 65
- acyclic (*graph utilities*), 43
- AIC.bn (*score*), 67
- AIC.bn.fit (*bn.fit utilities*), 22
- alarm, 8
- amat (*misc utilities*), 59
- amat<- (*misc utilities*), 59
- arc operations, 10
- arc.strength, 11, 25, 29, 32, 68
- arcs (*misc utilities*), 59
- arcs<- (*misc utilities*), 59
- as.bn (*model string utilities*), 61
- as.character.bn (*model string utilities*), 61
- asia, 13
- bn class, 14
- bn-class, 35, 37, 51, 57, 70
- bn-class (*bn class*), 14
- bn.boot, 16, 18, 29, 66
- bn.cv, 17, 17, 66
- bn.fit, 18, 19, 22, 23, 66
- bn.fit class, 19, 20, 22
- bn.fit plots, 20, 21
- bn.fit utilities, 20, 22
- bn.fit-class, 23
- bn.fit-class (*bn.fit class*), 20
- bn.fit.barchart (*bn.fit plots*), 21
- bn.fit.dnode (*bn.fit class*), 20
- bn.fit.dotplot (*bn.fit plots*), 21
- bn.fit.gnode (*bn.fit class*), 20
- bn.fit.histogram (*bn.fit plots*), 21
- bn.fit.qqplot (*bn.fit plots*), 21
- bn.fit.xyplot (*bn.fit plots*), 21
- bn.kcv class, 25
- bn.kcv-class, 18
- bn.kcv-class (*bn.kcv class*), 25
- bn.moments (*bn.var*), 26
- bn.strength (*bn.strength class*), 25
- bn.strength class, 12, 25
- bn.strength-class (*bn.strength class*), 25
- bn.var, 26
- bnlearn (*bnlearn-package*), 2
- bnlearn-package, 11, 16, 26, 28, 29, 31, 35, 50, 51, 57, 67, 69
- bnlearn-package, 2
- boot.strength, 11, 12, 28
- children (*misc utilities*), 59
- children<- (*misc utilities*), 59
- choose.direction, 12, 29, 32, 68
- ci.test, 30
- coef.bn.fit (*bn.fit utilities*), 22
- compare, 33
- constraint-based algorithms, 51, 57, 70
- constraint-based algorithms, 34
- coronary, 36
- cpdag, 37
- cpquery, 38
- deal integration, 39
- directed (*graph utilities*), 43
- directed.arcs (*misc utilities*), 59
- drop.arc (*arc operations*), 10
- empty.graph (*graph generation utilities*), 41
- fast.iamb, 2
- fast.iamb (*constraint-based algorithms*), 34
- fitted.bn.fit (*bn.fit utilities*), 22
- gaussian.test, 40
- graph generation utilities, 41
- graph utilities, 43
- graphviz.plot, 45, 65, 71
- gs, 2
- gs (*constraint-based algorithms*), 34
- hailfinder, 46
- hc, 3
- hc (*score-based algorithms*), 69
- hybrid algorithms, 36, 50, 57, 70
- iamb, 2
- iamb (*constraint-based algorithms*), 34

- insurance, [52](#)
- inter.iamb, [2](#)
- inter.iamb (*constraint-based algorithms*), [34](#)
- leaf.nodes (*misc utilities*), [59](#)
- learning.test, [54](#)
- lizards, [55](#)
- local discovery algorithms, [36](#), [51](#), [56](#), [70](#)
- logLik.bn (*score*), [67](#)
- logLik.bn.fit (*bn.fit utilities*), [22](#)
- marks, [58](#)
- mb (*misc utilities*), [59](#)
- misc utilities, [59](#)
- mmhc, [3](#)
- mmhc (*hybrid algorithms*), [50](#)
- mmpc, [3](#)
- mmpc (*local discovery algorithms*), [56](#)
- model string utilities, [61](#)
- model2network (*model string utilities*), [61](#)
- modelstring (*model string utilities*), [61](#)
- modelstring<- (*model string utilities*), [61](#)
- moral (*cpdag*), [37](#)
- nbr (*misc utilities*), [59](#)
- node ordering utilities, [63](#)
- node.ordering (*node ordering utilities*), [63](#)
- nodes (*misc utilities*), [59](#)
- nparams (*misc utilities*), [59](#)
- ordering2blacklist (*node ordering utilities*), [63](#)
- parents (*misc utilities*), [59](#)
- parents<- (*misc utilities*), [59](#)
- path (*graph utilities*), [43](#)
- pdag2dag, [19](#)
- pdag2dag (*graph utilities*), [43](#)
- plot.bn, [46](#), [64](#)
- predict.bn.fit (*bn.fit utilities*), [22](#)
- random.graph (*graph generation utilities*), [41](#)
- rbn, [17](#), [18](#), [65](#)
- residuals.bn.fit (*bn.fit utilities*), [22](#)
- reverse.arc (*arc operations*), [10](#)
- root.nodes (*misc utilities*), [59](#)
- rsmax2, [3](#)
- rsmax2 (*hybrid algorithms*), [50](#)
- score, [11](#), [12](#), [19](#), [29](#), [51](#), [67](#), [69](#)
- score-based algorithms, [36](#), [51](#), [57](#)
- score-based algorithms, [69](#)
- set.arc, [19](#)
- set.arc (*arc operations*), [10](#)
- shd (*compare*), [33](#)
- skeleton (*graph utilities*), [43](#)
- snow integration, [3](#), [35](#), [56](#), [70](#)
- strength.plot, [25](#), [71](#)
- tabu, [3](#)
- tabu (*score-based algorithms*), [69](#)
- undirected.arcs (*misc utilities*), [59](#)
- vstructs (*cpdag*), [37](#)