

# Package ‘bnlearn’

November 27, 2015

**Type** Package

**Title** Bayesian Network Structure Learning, Parameter Learning and Inference

**Version** 3.9

**Date** 2015-11-27

**Depends** R (>= 2.14.0), methods

**Suggests** parallel, graph, Rgraphviz, lattice, gRain

**Author** Marco Scutari

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

**Description** Bayesian network structure learning, parameter learning and inference.

This package implements constraint-based (GS, IAMB, Inter-IAMB, Fast-IAMB, MMPC, Hiton-PC), pairwise (ARACNE and Chow-Liu), score-based (Hill-Climbing and Tabu Search) and hybrid (MMHC and RSMAX2) structure learning algorithms for discrete, Gaussian and conditional Gaussian networks, along with many score functions and conditional independence tests.

The Naive Bayes and the Tree-Augmented Naive Bayes (TAN) classifiers are also implemented.

Some utility functions (model comparison and manipulation, random data generation, arc orientation testing, simple and advanced plots) are included, as well as support for parameter estimation (maximum likelihood and Bayesian) and inference, conditional probability queries and cross-validation. Development snapshots with the latest bugfixes are available from [www.bnlearn.com](http://www.bnlearn.com).

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

**License** GPL (>= 2)

**LazyData** yes

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2015-11-27 14:59:41

**R topics documented:**

bnlearn-package . . . . .	3
alarm . . . . .	9
arc operations . . . . .	11
arc.strength . . . . .	13
asia . . . . .	16
bn class . . . . .	17
bn.boot . . . . .	18
bn.cv . . . . .	20
bn.fit . . . . .	23
bn.fit class . . . . .	26
bn.fit plots . . . . .	27
bn.fit utilities . . . . .	28
bn.kcv class . . . . .	31
bn.strength class . . . . .	32
choose.direction . . . . .	33
ci.test . . . . .	34
clgaussian.test . . . . .	36
compare . . . . .	37
configs . . . . .	38
constraint-based algorithms . . . . .	39
coronary . . . . .	41
cpdag . . . . .	42
cpquery . . . . .	43
deal integration . . . . .	46
dsep . . . . .	47
foreign files utilities . . . . .	48
gaussian.test . . . . .	50
gRain integration . . . . .	50
graph generation utilities . . . . .	51
graph integration . . . . .	53
graph utilities . . . . .	54
graphviz.plot . . . . .	55
halffinder . . . . .	57
hybrid algorithms . . . . .	60
insurance . . . . .	62
learning.test . . . . .	64
lizards . . . . .	65
local discovery algorithms . . . . .	66
marks . . . . .	67
misc utilities . . . . .	68
model string utilities . . . . .	71
naive.bayes . . . . .	72
node ordering utilities . . . . .	75
parallel integration . . . . .	76
plot.bn . . . . .	77
plot.bn.strength . . . . .	78

preprocess . . . . .	79
rbn . . . . .	81
relevant . . . . .	82
score . . . . .	83
score-based algorithms . . . . .	86
single-node local discovery . . . . .	87
strength.plot . . . . .	88
test counter . . . . .	90

**Index 92**

---

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

---

**Description**

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

**Details**

Package: bnlearn  
 Type: Package  
 Version: 3.9  
 Date: 2015-11-27  
 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 used in a structure learning algorithm.

- *Incremental Association* (`iamb`): based on the Markov blanket detection algorithm of the same name, 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.
- *Interleaved Incremental Association* (`inter.iamb`): another variant of IAMB which uses forward stepwise selection 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 initialize the learning process of each node.
- 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 `parallel` package. See [parallel integration](#) for a sample usage.

The computational complexity of these algorithms is polynomial in the number of tests, usually  $O(N^2)$  (but super-exponential 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 *(partial) correlation graph*. Therefore all the arcs are undirected, and no attempt is made to detect their orientation. They are often used in hybrid learning algorithms.

- *Max-Min Parents and Children* (`mmpc`): a forward selection technique for neighbourhood detection based on the maximization of the minimum association measure observed with any subset of the nodes selected in the previous iterations.

- *Hiton Parents and Children* ([si.hiton.pc](#)): a fast forward selection technique for neighbourhood detection designed to exclude nodes early based on the marginal association. The implementation follows the Semi-Interleaved variant of the algorithm.
- *Chow-Liu* ([chow.liu](#)): an application of the minimum-weight spanning tree and the information inequality. It learns the tree structure closest to the true one in the probability space.
- *ARACNE* ([aracne](#)): an improved version of the Chow-Liu algorithm that is able to learn polytrees.

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

### Bayesian Network classifiers

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

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

### Available (conditional) independence tests

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

- *discrete case* (categorical variables)
  - *mutual information*: an information-theoretic distance measure. It's proportional to the log-likelihood ratio (they differ by a  $2n$  factor) and is related to the deviance of the tested models. The asymptotic  $\chi^2$  test (`mi` and `mi-adj`, with adjusted degrees of freedom), the Monte Carlo permutation test (`mc-mi`), the sequential Monte Carlo permutation test (`smc-mi`), and the semiparametric test (`sp-mi`) are implemented.
  - *shrinkage estimator* for the *mutual information* (`mi-sh`): an improved asymptotic  $\chi^2$  test based on the James-Stein estimator for the mutual information.
  - *Pearson's  $X^2$* : the classical Pearson's  $X^2$  test for contingency tables. The asymptotic  $\chi^2$  test (`x2` and `x2-adj`, with adjusted degrees of freedom), the Monte Carlo permutation test (`mc-x2`), the sequential Monte Carlo permutation test (`smc-x2`) and semiparametric test (`sp-x2`) are implemented.
- *discrete case* (ordered factors)
  - *Jonckheere-Terpstra*: a trend test for ordinal variables. The asymptotic normal test (`jt`), the Monte Carlo permutation test (`mc-jt`) and the sequential Monte Carlo permutation test (`smc-jt`) are implemented.
- *continuous case* (normal variables)
  - *linear correlation*: Pearson's linear correlation. The exact Student's t test (`cor`), the Monte Carlo permutation test (`mc-cor`) and the sequential Monte Carlo permutation test (`smc-cor`) are implemented.

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

### Available network scores

Available scores (and the respective labels) are:

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

### 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.

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

### Error detection and correction: the strict mode

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

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

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

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

Each correction results in a warning.

### Author(s)

Marco Scutari  
UCL Genetics Institute (UGI)  
University College London

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

### References

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

Nagarajan R, Scutari M, Lebre S (2013). "Bayesian Networks in R with Applications in Systems Biology". Springer.

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

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

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

Pearl J (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

## Examples

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

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

## 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
res3 = gs(learning.test, blacklist = blacklist)
plot(res3)
# force E - F direction (E -> F).
whitelist = data.frame(from = c("E"), to = c("F"))
```



```

whitelist
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()
# if something seems wrong, try the unoptimized version
# in strict mode (inconsistencies trigger errors):
res = gs(learning.test, optimized = FALSE, strict = TRUE, debug = TRUE)
# or disable strict mode to let the algorithm fix errors on the fly:
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.
- AC02 (*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 complete BN can be downloaded from <http://www.bnlearn.com/bnrepository>.

**Source**

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

Elidan G (2001). "Bayesian Network Repository".

<http://www.cs.huji.ac.il/site/labs/compbio/Repository>.

**Examples**

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

---

arc operations

*Drop, add or set the direction of an arc or an edge*

---

**Description**

Drop, add or set the direction of an arc or an edge.

**Usage**

```
# arc operations.
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)

# edge (i.e. undirected arc) operations
set.edge(x, from, to, check.cycles = TRUE, debug = FALSE)
drop.edge(x, from, to, debug = FALSE)
```

**Arguments**

x                    an object of class bn.  
 from                a character string, the label of a node.  
 to                   a character string, the label of another node.

<code>check.cycles</code>	a boolean value. If TRUE the graph is tested for acyclicity; otherwise the graph is returned anyway.
<code>debug</code>	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Details

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

- if there is no arc between `from` and `to`, the arc `from → to` is added.
- if there is an undirected arc between `from` and `to`, its direction is set to `from → to`.
- if the arc `to → from` is present, it is reversed.
- if the arc `from → 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 a directed or an undirected arc between `from` and `to`, it is 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 → from` is present, it is reversed.
- if the arc `from → to` is present, it is reversed.

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

- if there is no arc between `from` and `to`, the undirected arc `from - to` is added.
- if there is an undirected arc between `from` and `to`, no action is taken.
- if either the arc `from → to` or the arc `to → from` are present, they are replaced with the undirected arc `from - to`.

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

- if there is no undirected arc between `from` and `to`, no action is taken.
- if there is an undirected arc between `from` and `to`, it is removed.
- if there is a directed arc between `from` and `to`, no action is taken.

### Value

All functions return invisibly an updated copy of `x`.

### Author(s)

Marco Scutari

**Examples**

```

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

## use debug = TRUE to get more information.
set.arc(res, "A", "B")
drop.arc(res, "A", "B")
drop.edge(res, "A", "B")
reverse.arc(res, "A", "D")

```

---

arc.strength	<i>Measure arc strength</i>
--------------	-----------------------------

---

**Description**

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

**Usage**

```

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

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

```

**Arguments**

x	an object of class bn.
networks	a list, containing either object of class bn or arc sets (matrices or data frames with two columns, optionally labeled "from" and "to").
data	a data frame containing the data the Bayesian network was learned from.
cluster	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> for details and a simple example.
strength	an object of class bn.strength, see below.
threshold	a numeric value, the minimum strength required for an arc to be included in the averaged network. The default value is the threshold attribute of the strength argument.
nodes	a vector of character strings, the labels of the nodes in the network. In <a href="#">averaged.network</a> , it defaults to the set of the unique node labels in the strength argument.

crit <code>erion</code>	a character string, the label of a score function or an independence test. See <a href="#">bnlearn-package</a> for details.
R	a positive integer, the number of bootstrap replicates.
m	a positive integer, the size of each bootstrap replicate.
weights	a vector of non-negative numbers, to be used as weights when averaging network structures to compute strength coefficients. If NULL, weights are assumed to be uniform.
cpdag	a boolean value. If TRUE the (PDAG of) the equivalence class is used instead of the network structure itself. It should make it easier to identify score-equivalent arcs.
algorithm	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are <code>gs</code> , <code>iamb</code> , <code>fast.iamb</code> , <code>inter.iamb</code> , <code>mmpc</code> , <code>hc</code> , <code>tabu</code> , <code>mmhc</code> and <code>rsmax2</code> . See <a href="#">bnlearn-package</a> and the documentation of each algorithm for details.
algorithm.args	a list of extra arguments to be passed to the learning algorithm.
...	additional tuning parameters for the network score (if <code>crit<code>erion</code></code> is the label of a score function, see <a href="#">score</a> for details), the conditional independence test (currently the only one is B, the number of permutations).
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Details

`arc.strength` computes a measure of confidence or strength for each arc, while keeping fixed the rest of the network structure.

If `criterion` is a conditional independence test, the strength is a p-value (so the lower the value, the stronger the relationship). The conditional independence test would be that to drop the arc from the network. 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. In other words, it is the difference between the score of the network including the arc and the score of the network in which the arc is not present. Negative values correspond to decreases in the network score and positive values correspond to increases in the network score (the stronger the relationship, the more negative the difference). There may be additional parameters depending on the choice of the score, see [score](#) for details.

Model averaging is supported for objects of class `bn.strength` returned by [boot.strength](#) or by [custom.strength](#). The returned network contains the arcs whose strength is greater than the threshold attribute of the `bn.strength` object passed to `averaged.network`.

### Value

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

`averaged.network` returns an object of class `bn`.

See [bn.strength class](#) and [bn-class](#) for details.

**Note**

averaged.network typically returns a completely directed graph; an arc can be undirected if and only if the probability of each of its directions is exactly 0.5. This may happen, for example, if the arc is undirected in all the networks being averaged.

**Author(s)**

Marco Scutari

**References****for model averaging and bootstrap strength (confidence):**

Friedman N, Goldszmidt M, Wyner A (1999). "Data Analysis with Bayesian Networks: A Bootstrap Approach". In "UAI '99: Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence", pp. 196-201. Morgan Kaufmann.

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

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

**See Also**

[strength.plot](#), [choose.direction](#), [score](#), [ci.test](#).

**Examples**

```
data(learning.test)
res = gs(learning.test)
res = set.arc(res, "A", "B")
arc.strength(res, learning.test)

## Not run:
arcs = boot.strength(learning.test, algorithm = "hc")
arcs[(arcs$strength > 0.85) & (arcs$direction >= 0.5), ]
averaged.network(arcs)

start = random.graph(nodes = names(learning.test), num = 50)
netlist = lapply(start, function(net) {
  hc(learning.test, score = "bde", iss = 10, start = net) })
arcs = custom.strength(netlist, nodes = names(learning.test),
  cpdag = FALSE)
arcs[(arcs$strength > 0.85) & (arcs$direction >= 0.5), ]
modelstring(averaged.network(arcs))

## 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 complete BN can be downloaded from <http://www.bnlearn.com/bnrepository>.

### 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 S3 class bn.

## Details

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

- learning: a list containing some information about the results of the learning algorithm. It's never changed afterward.
  - whitelist: a sanitized copy of the whitelist parameter (a two-column matrix, whose columns are labeled from and to).
  - blacklist: a sanitized copy of the blacklist parameter (a two-column matrix, whose columns are labeled from and to).
  - test: the label of the conditional independence test used by the learning algorithm (a character string). The label of the network score is used for score-based and hybrid algorithms, and "none" for randomly generated graphs.
  - ntests: the number of conditional independence tests or score comparisons used in the learning (an integer value).
  - algo: the label of the learning algorithm or the random generation algorithm used to generate the network (a character string).
  - args: a list. The values of the parameters of either the conditional tests or the scores used in the learning process. Only the relevant ones are stored, so this may be an empty list.
    - \* alpha: the target nominal type I error rate (a numeric value) of the conditional independence tests.
    - \* iss: a positive numeric value, the imaginary sample size used by the bge and bde scores.
    - \* phi: a character string, either heckerman or bottcher; used by the bge score.
    - \* k: a positive numeric value, the penalty per parameter used by the aic, aic-g, bic and bic-g scores.
    - \* prob: the probability of each arc to be present in a graph generated by the ordered graph generation algorithm.
    - \* burn.in: the number of iterations for the ic-dag graph generation algorithm to converge to a stationary (and uniform) probability distribution.

- \* `max.degree`: the maximum degree for any node in a graph generated by the `ic-dag` graph generation algorithm.
  - \* `max.in.degree`: the maximum in-degree for any node in a graph generated by the `ic-dag` graph generation algorithm.
  - \* `max.out.degree`: the maximum out-degree for any node in a graph generated by the `ic-dag` graph generation algorithm.
  - \* `training`: a character string, the label of the training node in a Bayesian network classifier.
  - \* `threshold`: the threshold used to determine which arcs are significant when averaging network structures.
- `nodes`: a list. Each element is named after a node and contains the following elements:
    - `mb`: the Markov blanket of the node (a vector of character strings).
    - `nbr`: the neighbourhood of the node (a vector of character strings).
    - `parents`: the parents of the node (a vector of character strings).
    - `children`: the children of the node (a vector of character strings).
  - `arcs`: the arcs of the Bayesian network (a two-column matrix, whose columns are labeled from and to). Undirected arcs are stored as two directed arcs with opposite directions between the corresponding incident nodes.

Additional (optional) components under learning:

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

### Author(s)

Marco Scutari

---

bn.boot

*Parametric and nonparametric bootstrap of Bayesian networks*

---

### Description

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

## Usage

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

## Arguments

data	a data frame containing the variables in the model.
statistic	a function or a character string (the name of a function) to be applied to each bootstrap replicate.
R	a positive integer, the number of bootstrap replicates.
m	a positive integer, the size of each bootstrap replicate.
sim	a character string indicating the type of simulation required. Possible values are "ordinary" (the default) and "parametric".
algorithm	a character string, the learning algorithm to be applied to the bootstrap replicates. Possible values are <code>gs</code> , <code>iamb</code> , <code>fast.iamb</code> , <code>inter.iamb</code> , <code>mmpc</code> , <code>hc</code> , <code>tabu</code> , <code>mmhc</code> and <code>rsmax2</code> . See <a href="#">bnlearn-package</a> and documentation of each algorithm for details.
algorithm.args	a list of extra arguments to be passed to the learning algorithm.
statistic.args	a list of extra arguments to be passed to the function specified by <code>statistic</code> .
cluster	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> for details and a simple example.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

## Details

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

## Value

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

## Author(s)

Marco Scutari

## References

Friedman N, Goldszmidt M, Wyner A (1999). "Data Analysis with Bayesian Networks: A Bootstrap Approach". In "UAI '99: Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence", pp. 196-201. 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)

## End(Not run)
```

---

 bn.cv

---

*Cross-validation for Bayesian networks*


---

**Description**

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

**Usage**

```
bn.cv(data, bn, loss = NULL, k = 10, m, runs = 1, algorithm.args = list(),
      loss.args = list(), fit = "mle", fit.args = list(), method = "k-fold",
      cluster = NULL, debug = FALSE)

## S3 method for class 'bn.kcv'
plot(x, ..., main, xlab, ylab, connect = FALSE)
## S3 method for class 'bn.kcv.list'
plot(x, ..., main, xlab, ylab, connect = FALSE)
```

**Arguments**

data	a data frame containing the variables in the model.
bn	either a character string (the label of the learning algorithm to be applied to the training data in each iteration) or an object of class bn (a fixed network structure).
loss	a character string, the label of a loss function. If none is specified, the default loss function is the <i>Classification Error</i> for Bayesian networks classifiers; otherwise, 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.
m	a positive integer number, the size of the test set in hold-out cross-validation.
runs	a positive integer number, the number of times cross-validation will be run.
algorithm.args	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 <a href="#">bn.fit</a> for details.
<code>fit.args</code>	additional arguments for the parameter estimation procedure, see again <a href="#">bn.fit</a> for details.
<code>method</code>	a character string, either <code>k-fold</code> or <code>hold-out</code> . See below for details.
<code>cluster</code>	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> for details and a simple example.
<code>debug</code>	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.
<code>x</code>	an object of class <code>bn.kcv</code> or <code>bn.kcv.list</code> returned by <code>bn.cv</code> .
<code>...</code>	additional objects of class <code>bn.kcv</code> or <code>bn.kcv.list</code> to plot alongside the first.
<code>main, xlab, ylab</code>	the title of the plot, an array of labels for the boxplot, the label for the y axis.
<code>connect</code>	a logical value. If TRUE, the medians points in the boxplots will be connected by a segmented line.

### Value

An object of class `bn.kcv.list` if `runs` is at least 2, an object of class `bn.kcv` if `runs` is equal to 1.

### Cross-Validation Strategies

The following cross-validation strategies are implemented:

- *k-fold*: the data are split in  $k$  subsets of equal size. For each subset in turn, `bn` is fitted (and possibly learned as well) on the other  $k - 1$  subsets and the loss function is then computed using that subset. Loss estimates for each of the  $k$  subsets are then combined to give an overall loss for data.
- *hold-out*:  $k$  subsamples of size  $m$  are sampled independently without replacement from the data. For each subsample, `bn` is fitted (and possibly learned) on the remaining  $m - \text{nrow}(\text{data})$  samples and the loss function is computed on the  $m$  observations in the subsample. The overall loss estimate is the average of the  $k$  loss estimates from the subsamples.

If either cross-validation is used with multiple runs, the overall loss is the average of the loss estimates from the different runs.

### Loss Functions

The following loss functions are implemented:

- *Log-Likelihood Loss* (`logl`): also known as *negative entropy* or *negentropy*, it is 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 in a discrete network. Frequentist predictions are used, so the values of the target node are predicted using only the information present in its local distribution (from its parents).
- *Posterior Classification Error* (pred-lw): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates.
- *Predictive Correlation* (cor): the *correlation* between the observed and the predicted values for a single node in a Gaussian Bayesian network.
- *Posterior Predictive Correlation* (cor-lw): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates.
- *Mean Squared Error* (mse): the *mean squared error* between the observed and the predicted values for a single node in a Gaussian Bayesian network.
- *Posterior Mean Squared Error* (mse-lw): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates.

Optional arguments that can be specified in `loss.args` are:

- `target`: a character string, the label of target node for prediction in all loss functions but `logl`, `logl-g` and `logl-cg`.
- `from`: a vector of character strings, the labels of the nodes used to predict the target node in `pred-lw`, `cor-lw` and `mse-lw`. The default is to use all the other nodes in the network. Loss functions `pred`, `cor` and `mse` implicitly predict only from the parents of the target node.
- `n`: a positive integer, the number of particles used by likelihood weighting for `pred-lw`, `cor-lw` and `mse-lw`. The default value is 500.

Note that if `bn` is a Bayesian network classifier, `pred` and `pred-lw` both give exact posterior predictions computed using the closed-form formulas for naive Bayes and TAN.

### Plotting Results from Cross-Validation

Both plot methods accept any combination of objects of class `bn.kcv` or `bn.kcv.list` (the first as the `x` argument, the remaining as the `...` argument) and plot the respected expected loss values side by side. For a `bn.kcv` object, this mean a single point; for a `bn.kcv.list` object this means a boxplot.

### 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"))
bn.cv(gaussian.test, 'mmhc', method = "hold-out", k = 5, m = 50, runs = 2)

gaussian.subset = gaussian.test[1:50, ]
cv.gs = bn.cv(gaussian.subset, 'gs', runs = 10)
cv.iamb = bn.cv(gaussian.subset, 'iamb', runs = 10)
cv.inter = bn.cv(gaussian.subset, 'inter.iamb', runs = 10)
plot(cv.gs, cv.iamb, cv.inter,
     xlab = c("Grow-Shrink", "IAMB", "Inter-IAMB"), connect = TRUE)
```

---

 bn.fit

*Fit the parameters of a Bayesian network*


---

## Description

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

## Usage

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

## Arguments

x	an object of class bn (for bn.fit and custom.fit) or an object of class bn.fit (for bn.net).
data	a data frame containing the variables in the model.
dist	a named list, with element for each node of x. See below.
method	a character string, either mle for <i>Maximum Likelihood parameter estimation</i> or bayes for <i>Bayesian parameter estimation</i> (currently implemented only for discrete data).
...	additional arguments for the parameter estimation procedure, see below.
ordinal	a vector of character strings, the labels of the discrete nodes which should be saved as ordinal random variables (bn.fit.onode) instead of unordered factors (bn.fit.dnode).
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

## Details

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

An in-place replacement method is available to change the parameters of each node in a `bn.fit` object; see the examples for discrete, continuous and hybrid networks below. For a discrete node (class `bn.fit.dnode` or `bn.fit.onode`), the new parameters must be in a table object. For a Gaussian node (class `bn.fit.gnode`), the new parameters can be defined either by an `lm`, `glm` or `penlim` object (the latter is from the `penalized` package) or in a list with elements named `coef`, `sd` and optionally `fitted` and `resid`. For a conditional Gaussian node (class `bn.fit.cnode`), the new parameters can be defined by a list with elements named `coef`, `sd` and optionally `fitted`, `resid` and `configs`. In both cases `coef` should contain the new regression coefficients, `sd` the standard deviation of the residuals, `fitted` the fitted values and `resid` the residuals. `configs` should contain the configurations if the discrete parents of the conditional Gaussian node, stored as a factor.

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

## Value

`bn.fit` returns an object of class `bn.fit`, `bn.net` an object of class `bn`. See [bn class](#) and [bn.fit class](#) for details.

## Note

Due to the way Bayesian networks are defined it is 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).

The conditional probabilities in the local distributions of discrete nodes have a maximum likelihood estimate of NaN for all parents configurations that are not observed in data. Such missing values propagate to the results of functions such as `predict`. Bayesian posterior estimates do not have this problem, and are very close to the corresponding maximum likelihood estimates when `iss` is small.

## Author(s)

Marco Scutari

## See Also

[bn.fit utilities](#), [bn.fit plots](#).



**Examples**

```

data(learning.test)

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

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

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

# Gaussian Bayesian network from expert knowledge.
distA = list(coef = c("(Intercept)" = 2), sd = 1)
distB = list(coef = c("(Intercept)" = 1), sd = 1.5)
distC = list(coef = c("(Intercept)" = 0.5, "A" = 0.75, "B" = 1.32), sd = 0.4)
cfit = custom.fit(net, dist = list(A = distA, B = distB, C = distC))

# conditional Gaussian Bayesian network from expert knowledge.
cptA = matrix(c(0.4, 0.6), ncol = 2, dimnames = list(NULL, c("LOW", "HIGH")))
distB = list(coef = c("(Intercept)" = 1), sd = 1.5)
distC = list(coef = matrix(c(1.2, 2.3, 3.4, 4.5), ncol = 2,
                          dimnames = list(c("(Intercept)", "B"), NULL)),
            sd = c(0.3, 0.6))
cgfit = custom.fit(net, dist = list(A = cptA, B = distB, C = distC))

```

bn.fit class

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

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

**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), the object also has class `bn.fit.dnet`; each node has class `bn.fit.dnode` and contains the following elements:

- `node`: a character string, the label of the node.
- `parents`: a vector of character strings, the labels of the parents of the node.
- `children`: a vector of character strings, the labels of the children of the node.
- `prob`: a (multi)dimensional numeric table, the conditional probability table of the node given its parents.

Nodes encoding ordinal variables (i.e. ordered factors) have class `bn.fit.onode` and contain the same elements as `bn.fit.dnode` nodes. Networks containing only ordinal nodes also have class `bn.fit.onet`, while those containing both ordinal and multinomial nodes also have class `bn.fit.donet`.

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

- `node`: a character string, the label of the node.
- `parents`: a vector of character strings, the labels of the parents of the node.
- `children`: a vector of character strings, the labels of the children of the node.
- `coefficients`: a numeric vector, the linear regression coefficients of the parents against the node.
- `residuals`: a numeric vector, the residuals of the linear regression.
- `fitted.values`: a numeric vector, the fitted mean values of the linear regression.
- `sd`: a numeric value, the standard deviation of the residuals (i.e. the standard error).

Hybrid (i.e. conditional linear Gaussian) networks also have class `bn.fit.gnet`. Gaussian nodes have class `bn.fit.gnode`, discrete nodes have class `bn.fit.dnode` and conditional Gaussian nodes have class `bn.fit.cgnode`. Each node contains the following elements:

- `node`: a character string, the label of the node.
- `parents`: a vector of character strings, the labels of the parents of the node.
- `children`: a vector of character strings, the labels of the children of the node.
- `dparents`: an integer vector, the indexes of the discrete parents in parents.

- `gparents`: an integer vector, the indexes of the continuous parents in parents.
- `dlevels`: a list containing the levels of the discrete parents in parents.
- `coefficients`: a numeric matrix, the linear regression coefficients of the continuous parents. Each column corresponds to a configuration of the discrete parents.
- `residuals`: a numeric vector, the residuals of the linear regression.
- `fitted.values`: a numeric vector, the fitted mean values of the linear regression.
- `configs`: an integer vector, the indexes of the configurations of the discrete parents.
- `sd`: a numeric vector, the standard deviation of the residuals (i.e. the standard error) for each configuration of the discrete parents.

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

### Author(s)

Marco Scutari

---

bn.fit plots

*Plot fitted Bayesian networks*

---

### Description

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

### Usage

```
## for Gaussian Bayesian networks.
bn.fit.qqplot(fitted, xlab = "Theoretical Quantiles",
  ylab = "Sample Quantiles", main = "Normal Q-Q Plot", ...)
bn.fit.histogram(fitted, density = TRUE, xlab = "Residuals",
  ylab = ifelse(density, "Density", ""),
  main = "Histogram of the residuals", ...)
bn.fit.xyplot(fitted, xlab = "Fitted values",
  ylab = "Residuals", main = "Residuals vs Fitted", ...)
## for discrete (multinomial and ordinal) Bayesian networks.
bn.fit.barchart(fitted, xlab = "Probabilities",
  ylab = "Levels", main = "Conditional Probabilities", ...)
bn.fit.dotplot(fitted, xlab = "Probabilities",
  ylab = "Levels", main = "Conditional Probabilities", ...)
```

**Arguments**

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

**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'
sigma(object, ...)
## S3 method for class 'bn.fit'
predict(object, node, data, method = "parents", ..., debug = FALSE)
## S3 method for class 'bn.fit'
logLik(object, data, nodes, by.sample = FALSE, ...)
## S3 method for class 'bn.fit'
AIC(object, data, ..., k = 1)
## S3 method for class 'bn.fit'
BIC(object, data, ...)

## methods available for "bn.fit.dnode"
## S3 method for class 'bn.fit.dnode'
coef(object, ...)

## methods available for "bn.fit.onode"
## S3 method for class 'bn.fit.onode'
coef(object, ...)

## 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'
sigma(object, ...)

## methods available for "bn.fit.cgnode"
## S3 method for class 'bn.fit.cgnode'
fitted(object, ...)
## S3 method for class 'bn.fit.cgnode'
coef(object, ...)
## S3 method for class 'bn.fit.cgnode'
residuals(object, ...)
## S3 method for class 'bn.fit.cgnode'
sigma(object, ...)

```

### Arguments

object	an object of class <code>bn.fit</code> , <code>bn.fit.dnode</code> , <code>bn.fit.gnode</code> , <code>bn.fit.cgnode</code> or <code>bn.fit.onode</code> .
node	a character string, the label of a node.
nodes	a vector of character strings, the label of a nodes whose log-likelihood components are to be computed.
data	a data frame containing the variables in the model.

method	a character string, the method used to estimate predictions. See below.
...	additional arguments. See below.
k	a numeric value, the penalty per parameter to be used; the default $k = 1$ gives the expression used to compute AIC.
by.sample	a boolean value. If TRUE, logLik returns a vector containing the the log-likelihood of each observations in the sample. If FALSE, logLik returns a single value, the likelihood of the whole sample.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Details

coef (and its alias coefficients) extracts model coefficients (which are conditional probabilities for discrete nodes and linear regression coefficients for Gaussian and conditional Gaussian nodes).

residuals (and its alias resid) extracts model residuals and fitted (and its alias fitted.values) extracts fitted values from Gaussian and conditional Gaussian nodes. If the bn.fit object does not include the residuals or the fitted values for the node of interest both functions return NULL.

sigma extracts the standard deviations of the residuals from Gaussian and conditional Gaussian networks and nodes.

predict returns the predicted values for node given the data specified by data and the fitted network. Depending on the value of method, the predicted values are computed as follows.

- parents: the predicted values are computed by plugging in the new values for the parents of node in the local probability distribution of node extracted from fitted.
- bayes-lw: the predicted values are computed by averaging likelihood weighting simulations performed using all the available nodes as evidence (obviously, with the exception of the node whose values we are predicting). The number of random samples which are averaged for each new observation is controlled by the n optional argument; the default is 500. If the variable being predicted is discrete, the predicted level is that with the highest conditional probability. If the variable is continuous, the predicted value is the expected value of the conditional distribution.

### Value

predict returns a numeric vector (for Gaussian and conditional Gaussian nodes), a factor (for categorical nodes) or an ordered factor (for ordinal nodes).

logLik returns a numeric vector or a single numeric value, depending on the value of by.sample. AIC and BIC always return a single numeric value.

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 or matrix (if object has class bn.fit.dnode, bn.fit.gnode, bn.fit.cnode or bn.fit.onode).

### Note

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

predict accepts either a bn or a bn.fit object as its first argument. For the former, the parameters of the network are fitted on data, that is, the observations whose class labels the function is trying to predict.

### 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)
coefficients(fitted$C)
str(residuals(fitted))

data(learning.test)
res2 = hc(learning.test)
fitted2 = bn.fit(res2, learning.test)
coefficients(fitted2$E)
```

---

bn.kcv class

*The bn.kcv class structure*

---

### Description

The structure of an object of S3 class bn.kcv or bn.kcv.list.

### Details

An object of class bn.kcv.list is a list whose elements are objects of class bn.kcv.

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

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

If the loss function requires to predict values from the test sets, each element also contains:

- predicted: a factor or a numeric vector, the predicted values for the target node in the test set.
- observed: a factor or a numeric vector, the observed values for the target node in the test set.

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

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

### Author(s)

Marco Scutari

---

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

---

### Description

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

### Details

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

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

and some additional attributes:

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

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

Only the `plot` method is defined for this class; therefore, it can be manipulated as a standard data frame.

### Author(s)

Marco Scutari



---

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

---

### Description

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

### Usage

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

### Arguments

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

### Details

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

### Value

choose.direction returns invisibly an updated copy of x.

### Author(s)

Marco Scutari

### See Also

[score](#), [arc.strength](#).

**Examples**

```

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

## the arc A - B has no direction.
choose.direction(res, learning.test, arc = c("A", "B"), debug = TRUE)

## let's see score equivalence in action.
choose.direction(res, learning.test, criterion = "aic",
  arc = c("A", "B"), debug = TRUE)

## 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)

## Not run:
choose.direction(res, learning.test, arc = c("D", "E"), criterion = "bootstrap",
  R = 100, algorithm = "iamb", algorithm.args = list(test = "x2"), cpdag = TRUE,
  debug = TRUE)

## End(Not run)

```

---

ci.test

*Independence and Conditional Independence Tests*


---

**Description**

Perform either an independence test or a conditional independence test.

**Usage**

```
ci.test(x, y, z, data, test, B, debug = FALSE)
```

**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 categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> for details.

B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the test argument is not a permutation test.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Value

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

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

### Author(s)

Marco Scutari

### References

#### for parametric and discrete permutation tests:

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

#### for shrinkage tests:

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

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

#### for continuous permutation tests:

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

#### for semiparametric discrete tests:

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

### See Also

[choose.direction](#), [arc.strength](#).

## Examples

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

# using a data frame and column labels.
ci.test(x = "F" , y = "B", z = c("C", "D"), data = gaussian.test)
# using a data frame.
ci.test(gaussian.test)
# using factor objects.
attach(learning.test)
ci.test(x = F , y = B, z = data.frame(C, D))
```

---

clgaussian.test	<i>Synthetic (mixed) data set to test learning algorithms</i>
-----------------	---

---

## Description

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

## Usage

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

## Format

The `clgaussian.test` data set contains one normal (Gaussian) variable, 4 discrete variables and 3 conditional 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(clgaussian.test)
res = empty.graph(names(clgaussian.test))
modelstring(res) = "[A][B][C][H][D|A:H][F|B:C][E|B:D][G|A:D:E:F]"
plot(res)
```

---

 compare

---

*Compare two different Bayesian networks*


---

### Description

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

### Usage

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

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

### Arguments

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

### Value

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

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

shd and hamming return a non-negative integer number.

### Note

Note that SHD, as defined in the reference, is defined on CPDAGs; therefore cpdag is called on both learned and true before computing the distance.

### Author(s)

Marco Scutari

## References

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

## Examples

```
data(learning.test)

e1 = model2network("[A][B][C|A:B][D|B][E|C][F|A:E]")
e2 = model2network("[A][B][C|A:B][D|B][E|C:F][F|A]")
shd(e2, e1, debug = TRUE)
unlist(compare(e1,e2))
compare(target = e1, current = e2, arcs = TRUE)
```

---

configs

*Construct configurations of discrete variables*

---

## Description

Create configurations of discrete variables, which can be used in modelling conditional probability tables.

## Usage

```
configs(data, all = TRUE)
```

## Arguments

data	a data frame containing factor columns.
all	a boolean value. If TRUE all configuration are included as levels in the return value; otherwise only configurations which are actually observed are considered.

## Details

discretize takes a data frame of continuous variables as its first argument and returns a second data frame of discrete variables, transformed using of three methods: interval, quantile or hartemink.

dedup screens the data for pairs of highly correlated variables, and discards one in each pair.

## Value

A factor with one element for each row of data, and levels as specified by all.

## Author(s)

Marco Scutari

**Examples**

```
data(learning.test)
configs(learning.test, all = TRUE)
configs(learning.test, all = FALSE)
```

---

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**

<code>x</code>	a data frame containing the variables in the model.
<code>cluster</code>	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> 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 categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> for details.
<code>alpha</code>	a numeric value, the target nominal type I error rate.

B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the <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.
optimized	a boolean value. See <a href="#">bnlearn-package</a> for details.
strict	a boolean value. If TRUE conflicting results in the learning process generate an error; otherwise they result in a warning.
undirected	a boolean value. If TRUE no attempt will be made to determine the orientation of the arcs; the returned (undirected) graph will represent the underlying structure of the Bayesian network.

**Value**

An object of class `bn`. See [bn-class](#) for details.

**Author(s)**

Marco Scutari

**References****for Grow-Shrink (GS):**

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

**for Incremental Association (IAMB):**

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

**for Fast-IAMB and Inter-IAMB:**

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

**See Also**

[local discovery algorithms](#), [score-based algorithms](#), [hybrid algorithms](#).



---

 coronary

*Coronary Heart Disease data set*


---

**Description**

Probable risk factors for coronary 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

*Equivalence classes, moral graphs and consistent extensions*


---

## Description

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

## Usage

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

## Arguments

x	an object of class bn.
arcs	a boolean value. If TRUE the arcs that are part of at least one v-structure are returned instead of the v-structures themselves.
moral	a boolean value. If TRUE we define a v-structure as in Pearl (2000); if FALSE, as in Koller and Friedman (2009). See below.
strict	a boolean value. If no consistent extension is possible and strict is TRUE, an error is generated; otherwise a partially extended graph is returned with a warning.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

## Details

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

Setting `moral` to FALSE in `cpdag` and `vstructs` makes those functions follow the definition from Koller and Friedman (2009); the default value of TRUE, on the other hand, makes those functions follow the definition from Pearl (2000). The former call *v-structures* both shielded and unshielded colliders (respectively *moral v-structures* and *immoral v-structures*); the latter requires v-structures to be unshielded colliders.

**Value**

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

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

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

**Author(s)**

Marco Scutari

**References**

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

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

Pearl J (2009). *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition.

**Examples**

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

---

cpquery

*Perform conditional probability queries*

---

**Description**

Perform conditional probability queries (CPQs).

**Usage**

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

mutilated(x, evidence)
```

### Arguments

fitted	an object of class <code>bn.fit</code> .
x	an object of class <code>bn</code> or <code>bn.fit</code> .
event, evidence	see below.
nodes	a vector of character strings, the labels of the nodes whose conditional distribution we are interested in.
cluster	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> for details and a simple example.
method	a character string, the method used to perform the conditional probability query. Currently only <i>logic sampling</i> ( <code>ls</code> , the default) and <i>likelihood weighting</i> ( <code>lw</code> ) are 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.

### Details

`cpquery` estimates the conditional probability of event given evidence using the method specified in the `method` argument.

`cpdist` generates random observations conditional on the evidence using the method specified in the `method` argument.

`mutilated` constructs the mutilated network used for sampling in *likelihood weighting*.

Note that both `cpquery` and `cpdist` are based on Monte Carlo particle filters, and therefore they may return slightly different values on different runs.

### Value

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

`cpdist` returns a data frame containing the observations generated from the conditional distribution of the nodes conditional on evidence. The data frame has class `c("bn.cpdist", "data.frame")`, and a `method` attribute storing the value of the `method` argument. In the case of *likelihood weighting*, the weights are also attached as an attribute called `weights`.

`mutilated` returns a `bn` or `bn.fit` object, depending on the class of `x`.

### 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 `event` or `evidence` is set to `TRUE` an unconditional probability query is performed with respect to that argument.

Three tuning parameters are available:

- `n`: a positive integer number, the number of random observations to generate from `fitted`. The default value is  $5000 * \log_{10}(\text{nparams.fitted}(fitted))$  for discrete and conditional Gaussian networks and  $500 * \text{nparams.fitted}(fitted)$  for Gaussian networks.
- `batch`: a positive integer number, the size of each batch of random observations. Defaults to  $10^4$ .
- `query.nodes`: a vector of character strings, the labels of the nodes involved in event and evidence. Simple queries do not require to generate observations from all the nodes in the network, so `cpquery` and `cpdist` try to identify which nodes are used in event and evidence and reduce the network to their upper closure. `query.nodes` may be used to manually specify these nodes when automatic identification fails; there is no reason to use it otherwise.

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

### Likelihood Weighting

The event argument must be an expression describing the event of interest, as in logic sampling. The evidence argument must be a named list:

- Each element corresponds to one node in the network and must contain the value that node will be set to when sampling.
- In the case of a continuous node, two values can also be provided. In that case, the value for that node will be sampled from a uniform distribution on the interval delimited by the specified values.
- In the case of a discrete or ordinal node, two or more values can also be provided. In that case, the value for that node will be sampled with uniform probability from the set of specified values.

If either event or evidence is set to TRUE an unconditional probability query is performed with respect to that argument.

Tuning parameters are the same as for logic sampling: `n`, `batch` and `query.nodes`.

Note that the observations returned by `cpdist` are generated from the mutilated network, and need to be weighted appropriately when computing summary statistics (for more details, see the references below). `cpquery` does that automatically when computing the final conditional probability. Also note that the `batch` argument is ignored in `cpdist` for speed and memory efficiency.

### Author(s)

Marco Scutari

### References

- Koller D, Friedman N (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.

**Examples**

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

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

---

deal integration

*bnlearn - deal package integration*


---

**Description**

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

**Export a bn object to deal**

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

```
# load the deal package.
> library(deal)
```

```
Attaching package: 'deal'
```

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

```
  modelstring,
  nodes,
  score
```

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

### Import a network structure from deal

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

### Author(s)

Marco Scutari

---

dsep

*Test d-separation*

---

### Description

Check whether two nodes are d-separated.

### Usage

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

**Arguments**

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

**Value**

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

**Author(s)**

Marco Scutari

**References**

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

**Examples**

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

---

foreign files utilities

*Read and write BIF, NET, DSC and DOT files*

---

**Description**

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

**Usage**

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

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

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



```
write.net(file, fitted)

# Graphviz DOT format.
write.dot(file, graph)
```

### Arguments

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

### Value

`read.bif`, `read.dsc` and `read.net` return an object of class `bn.fit`.  
`write.bif`, `write.dsc`, `write.net` and `write.dot` return NULL invisibly.

### Note

All the networks present in the Bayesian Network Repository have associated BIF, DSC and NET files that can be imported with `read.bif`, `read.dsc` and `read.net`.

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

DOT files can be read by Graphviz, Gephi and a variety of other programs.

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

### Author(s)

Marco Scutari

### References

Bayesian Network Repository, <http://www.bnlearn.com/bnrepository>.

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

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

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

Graphviz, <http://www.graphviz.org>.

Gephi, <http://gephi.org>.

---

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

---

**Description**

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

**Usage**

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

**Format**

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

**Note**

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

**Examples**

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

---

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

---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

**Note**

Conditional probability tables in `grain` objects must be completely specified; on the other hand, `bn.fit` allows NaN values for unobserved parents' configurations. Such `bn.fit` objects will be converted to `grain` objects by replacing the missing conditional probability distributions with uniform distributions.

Another solution to this problem is to fit another `bn.fit` with `method = "bayes"` and a low `iss` value, using the same data and network structure.

Ordinal nodes will be treated as categorical by `as.grain`, disregarding the ordering of the levels.

**Author(s)**

Marco Scutari

**Examples**

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

---

graph generation utilities

*Generate empty or random graphs*

---

**Description**

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

**Usage**

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

**Arguments**

<code>nodes</code>	a vector of character strings, the labels of the nodes.
<code>num</code>	an integer, the number of graphs to be generated.
<code>method</code>	a character string, the label of a score. Possible values are <code>ordered</code> ( <i>full ordering</i> based generation), <code>ic-dag</code> (Ide's and Cozman's <i>Generating Multi-connected DAGs</i> algorithm), <code>melancon</code> (Melancon's and Philippe's <i>Uniform Random Acyclic Digraphs</i> algorithm) and <code>empty</code> (generates empty graphs).

... additional tuning parameters (see below).  
 debug a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent. Ignored in some generation methods.

### Details

Available graph generation algorithms are:

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

Additional parameters for the random.graph function are:

- prob: the probability of each arc to be present in a graph generated by the ordered algorithm. The default value is  $2 / (\text{length}(\text{nodes}) - 1)$ , 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  $6 * \text{length}(\text{nodes})^2$ .
- every: return only one graph every number of steps instead of all the graphs generated with ic-dag and melancon. Since both algorithms are based on Markov Chain Monte Carlo approaches, high values of every result in a more diverse set of networks. The default value is 1, i.e. to return all the networks that are generated.
- max.degree: the maximum degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.
- max.in.degree: the maximum in-degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.
- max.out.degree: the maximum out-degree for any node in a graph generated by the ic-dag and melancon algorithms. The default value is Inf.

### Value

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

### Author(s)

Marco Scutari

## References

- Ide JS, Cozman FG (2002). "Random Generation of Bayesian Networks". In "SBIA '02: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence", pp. 366-375. Springer-Verlag.
- Melancon G, Dutour I, Bousquet-Melou M (2001). "Random Generation of Directed Acyclic Graphs". *Electronic Notes in Discrete Mathematics*, **10**, 202-207.
- Melancon G, Philippe F (2004). "Generating Connected Acyclic Digraphs Uniformly at Random". *Information Processing Letters*, **90**(4), 209-213.

## Examples

```
empty.graph(LETTERS[1:8])
random.graph(LETTERS[1:8])
plot(random.graph(LETTERS[1:8], method = "ic-dag", max.in.degree = 2))
plot(random.graph(LETTERS[1:8]))
plot(random.graph(LETTERS[1:8], prob = 0.2))
```

---

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

---

## Description

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

## Usage

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

## Arguments

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

## Value

An object of the relevant class.

**Note**

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

**Author(s)**

Marco Scutari

**Examples**

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

---

graph utilities

*Utilities to manipulate graphs*

---

**Description**

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

**Usage**

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

**Arguments**

<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.

underlying.graph	a boolean value. If TRUE the underlying undirected graph is used instead of the (directed) one from the x parameter.
ordering	the labels of all the nodes in the graph; their order is the node ordering used to set the direction of undirected arcs.
nodes	the labels of the nodes that induce the subgraph.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Value

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

### Author(s)

Marco Scutari

### References

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

### Examples

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

acyclic(res)
directed(res)
res = pdag2dag(res, ordering = LETTERS[1:6])
res
directed(res)
skeleton(res)
```

---

graphviz.plot

*Advanced Bayesian network plots*

---

### Description

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

### Usage

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

**Arguments**

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

**Details**

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

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

and optionally one or more of the following formatting parameters:

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

**Value**

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

**Author(s)**

Marco Scutari

**See Also**

[plot.bn](#).



---

hailfinder

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


---

## Description

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

## Usage

```
data(hailfinder)
```

## Format

The hailfinder data set contains the following 56 variables:

- N07muVerMo (*10.7mu vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- SubjVertMo (*subjective judgment of vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- QGVertMotion (*quasigeostrophic vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- CombVerMo (*combined vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- AreaMesoALS (*area of meso-alpha*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- SatContMoist (*satellite contribution to moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- RaoContMoist (*reading at the forecast center for moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- CombMoisture (*combined moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- AreaMoDryAir (*area of moisture and adry air*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- VISCloudCov (*visible cloud cover*): a three-level factor with levels Cloudy, PC and Clear.
- IRCLOUDCover (*infrared cloud cover*): a three-level factor with levels Cloudy, PC and Clear.
- CombClouds (*combined cloud cover*): a three-level factor with levels Cloudy, PC and Clear.
- CldShadeOth (*cloud shading, other*): a three-level factor with levels Cloudy, PC and Clear.
- AMInstabMt (*AM instability in the mountains*): a three-level factor with levels None, Weak and Strong.
- InsInMt (*instability in the mountains*): a three-level factor with levels None, Weak and Strong.
- WndHodograph (*wind hodograph*): a four-level factor with levels DCVZFavor, StrongWest, Westerly and Other.

- OutflowFrMt (*outflow from mountains*): a three-level factor with levels None, Weak and Strong.
- MorningBound (*morning boundaries*): a three-level factor with levels None, Weak and Strong.
- Boundaries (*boundaries*): a three-level factor with levels None, Weak and Strong.
- CldShadeConv (*cloud shading, convection*): a three-level factor with levels None, Some and Marked.
- CompPlFcst (*composite plains forecast*): a three-level factor with levels IncCapDecIns, LittleChange and DecCapIncIns.
- CapChange (*capping change*): a three-level factor with levels Decreasing, LittleChange and Increasing.
- LoLevMoistAd (*low-level moisture advection*): a four-level factor with levels StrongPos, WeakPos, Neutral and Negative.
- InsChange (*instability change*): three-level factor with levels Decreasing, LittleChange and Increasing.
- MountainFcst (*mountains (region 1) forecast*): a three-level factor with levels XNIL, SIG and SVR.
- Date (*date*): a six-level factor with levels May15\_Jun14, Jun15\_Jul1, Jul2\_Jul15, Jul16\_Aug10, Aug11\_Aug20 and Aug20\_Sep15.
- Scenario (*scenario*): an eleven-level factor with levels A, B, C, D, E, F, G, H, I, J and K.
- ScenReIAMCIN (*scenario relevant to AM convective inhibition*): a two-level factor with levels AB and CThruK.
- MorningCIN (*morning convective inhibition*): a four-level factor with levels None, PartInhibit, Stifling and TotalInhibit.
- AMCINInScen (*AM convective inhibition in scenario*): a three-level factor with levels LessThanAve, Average and MoreThanAve.
- CapInScen (*capping withing scenario*): a three-level factor with levels LessThanAve, Average and MoreThanAve.
- ScenReIAMIns (*scenario relevant to AM instability*): a six-level factor with levels ABI, CDEJ, F, G, H and K.
- LIfr12ZDENSd (*LI from 12Z DEN sounding*): a four-level factor with levels LIGt0, N1GtLIGt\_4, N5GtLIGt\_8 and LILt\_8.
- AMDewptCalPl (*AM dewpoint calculations, plains*): a three-level factor with levels Instability, Neutral and Stability.
- AMInsWliScen (*AM instability within scenario*): a three-level factor with levels LessUnstable, Average and MoreUnstable.
- InsScIInScen (*instability scaling within scenario*): a three-level factor with levels LessUnstable, Average and MoreUnstable.
- ScenRe134 (*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 (*realtime 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 complete BN can be downloaded from <http://www.bnlearn.com/bnrepository>.

### Source

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

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

**Examples**

```

# load the data and build the correct network from the model string.
data(hailfinder)
res = empty.graph(names(hailfinder))
modelstring(res) = paste("[N07muVerMo][SubjVertMo][QGVertMotion][SatContMoist][RaoContMoist]",
  "[VISCloudCov][IRCloudCover][AMInstabMt][WndHodograph][MorningBound][LoLevMoistAd][Date]",
  "[MorningCIN][LIfr12ZDENSd][AMDewptCalPl][LatestCIN][LLIW]",
  "[CombVerMo|N07muVerMo:SubjVertMo:QGVertMotion][CombMoisture|SatContMoist:RaoContMoist]",
  "[CombClouds|VISCloudCov:IRCloudCover][Scenario|Date][CurPropConv|LatestCIN:LLIW]",
  "[AreaMesoALS|CombVerMo][ScenRelAMCIN|Scenario][ScenRelAMIns|Scenario][ScenRel34|Scenario]",
  "[ScnRelPlFcst|Scenario][Dewpoints|Scenario][LowLLapse|Scenario][MeanRH|Scenario]",
  "[MidLLapse|Scenario][MvmtFeatures|Scenario][RHRatio|Scenario][SfcWndShfDis|Scenario]",
  "[SynForcng|Scenario][TempDis|Scenario][WindAloft|Scenario][WindFieldMt|Scenario]",
  "[WindFieldPln|Scenario][AreaMoDryAir|AreaMesoALS:CombMoisture]",
  "[AMCINInScen|ScenRelAMCIN:MorningCIN][AMInsWliScen|ScenRelAMIns:LIfr12ZDENSd:AMDewptCalPl]",
  "[CldShadeOth|AreaMesoALS:AreaMoDryAir:CombClouds][InsInMt|CldShadeOth:AMInstabMt]",
  "[OutflowFrMt|InsInMt:WndHodograph][CldShadeConv|InsInMt:WndHodograph][MountainFcst|InsInMt]",
  "[Boundaries|WndHodograph:OutflowFrMt:MorningBound][N34StarFcst|ScenRel34:PlainsFcst]",
  "[CompPlFcst|AreaMesoALS:CldShadeOth:Boundaries:CldShadeConv][CapChange|CompPlFcst]",
  "[InsChange|CompPlFcst:LoLevMoistAd][CapInScen|CapChange:AMCINInScen]",
  "[InsSc1InScen|InsChange:AMInsWliScen][R5Fcst|MountainFcst:N34StarFcst]",
  "[PlainsFcst|CapInScen:InsSc1InScen:CurPropConv:ScnRelPlFcst]",
  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**

x 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>mmpc</code> . See <a href="#">bnlearn-package</a> 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. Possible values are <code>hc</code> and <code>tabu</code> . See <a href="#">bnlearn-package</a> 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 categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> 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 <a href="#">bnlearn-package</a> 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 <a href="#">score</a> 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 <a href="#">bnlearn-package</a> 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 complete BN can be downloaded from <http://www.bnlearn.com/bnrepository>.

### Source

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

Elidan G (2001). "Bayesian Network Repository".

<http://www.cs.huji.ac.il/site/labs/compbio/Repository>.

### Examples

```
# load the data and build the correct network from the model string.
data(insurance)
res = empty.graph(names(insurance))
modelstring(res) = paste("[Age][Mileage][SocioEcon|Age][GoodStudent|Age:SocioEcon]",
  "[RiskAversion|Age:SocioEcon][OtherCar|SocioEcon][VehicleYear|SocioEcon:RiskAversion]",
  "[MakeModel|SocioEcon:RiskAversion][SeniorTrain|Age:RiskAversion]",
  "[HomeBase|SocioEcon:RiskAversion][AntiTheft|SocioEcon:RiskAversion]",
```

```

"[RuggedAuto|VehicleYear:MakeModel][Antilock|VehicleYear:MakeModel]",
"[DrivingSkill|Age:SeniorTrain][CarValue|VehicleYear:MakeModel:Mileage]",
"[Airbag|VehicleYear:MakeModel][DrivQuality|RiskAversion:DrivingSkill]",
"[Theft|CarValue:HomeBase:AntiTheft][Cushioning|RuggedAuto:Airbag]",
"[DrivHist|RiskAversion:DrivingSkill][Accident|DrivQuality:Mileage:Antilock]",
"[ThisCarDam|RuggedAuto:Accident][OtherCarCost|RuggedAuto:Accident]",
"[MedCost|Age:Accident:Cushioning][ILiCost|Accident]",
"[ThisCarCost|ThisCarDam:Theft:CarValue][PropCost|ThisCarCost:OtherCarCost]",
sep = "")
## Not run:
# there are too many nodes for plot(), use graphviz.plot().
graphviz.plot(res)
## End(Not run)

```

---

learning.test

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


---

## Description

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

## Usage

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

## Format

The learning.test data set contains the following variables:

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

## Note

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

## Examples

```

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

```



---

lizards

*Lizards' perching behaviour data set*

---

### Description

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

### Usage

```
data(lizards)
```

### Format

The lizards data set contains the following variables:

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

### Source

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

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

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

### Examples

```
# load the data and build the correct network from the model string.
data(lizards)
res = empty.graph(names(lizards))
modelstring(res) = "[Species][Diameter|Species][Height|Species]"
plot(res)
table(lizards[, c(3,2,1)])

## Not run:
# This data set is useful as it offers nominal values for
# the conditional mutual information and X^2 tests.
ci.test("Height", "Diameter", "Species", test = "mi", data = lizards)
ci.test("Height", "Diameter", "Species", test = "x2", data = lizards)

## End(Not run)
```

---

 local discovery algorithms

*Local discovery structure learning algorithms*


---

## Description

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

## Usage

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

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

## Arguments

<code>x</code>	a data frame containing the variables in the model.
<code>cluster</code>	an optional cluster object from package <b>parallel</b> . See <a href="#">parallel integration</a> 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>mi</code>	a character string, the estimator used for the pairwise (i.e. unconditional) mutual information coefficients in the ARACNE and Chow-Liu algorithms. Possible values are <code>mi</code> (discrete mutual information) and <code>mi-g</code> (Gaussian mutual information).
<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 categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> 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 TRUE a lot of debugging output is printed; otherwise the function is completely silent.

optimized	a boolean value. See <a href="#">bnlearn-package</a> for details.
strict	a boolean value. If TRUE conflicting results in the learning process generate an error; otherwise they result in a warning.

**Value**

An object of class bn. See [bn-class](#) for details.

**Author(s)**

Marco Scutari

**References**

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

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

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

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

**See Also**

[constraint-based algorithms](#), [score-based algorithms](#), [hybrid algorithms](#).

---

marks

*Examination marks data set*

---

**Description**

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

**Usage**

data(marks)

**Format**

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

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

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

**Source**

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

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

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

**Examples**

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

---

misc utilities

*Miscellaneous utilities*

---

**Description**

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

**Usage**

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

```

root.nodes(x)
leaf.nodes(x)
nnodes(x)

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

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

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

## shared with the graph package.
# these used to be a simple nodes(x) function.
## S4 method for signature 'bn'
nodes(object)
## S4 method for signature 'bn.fit'
nodes(object)
# these used to be a simple degree(x, node) function.
## S4 method for signature 'bn'
degree(object, Nodes)
## S4 method for signature 'bn.fit'
degree(object, Nodes)
# re-label the nodes.
## S4 replacement method for signature 'bn'
nodes(object) <- value
## S4 replacement method for signature 'bn.fit'
nodes(object) <- value

```

### Arguments

<code>x, object</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> requires an object of class <code>bn</code> .
<code>node, Nodes</code>	a character string, the label of a node.

value	either a vector of character strings (for parents and children), an adjacency matrix (for amat) or a data frame with two columns (optionally labeled "from" and "to", for arcs).
data	a data frame containing the data the Bayesian network was learned from. It's only needed if x is an object of class bn.
ignore.cycles	a boolean value. If TRUE the returned network will not be checked for cycles.
effective	a boolean value. If TRUE the number of non-zero free parameters is returned, that is, the effective degrees of freedom of the network; otherwise the theoretical number of parameters is returned.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Details

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

### Value

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

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

narcs and nnodes return the number of arcs and nodes in the graph, respectively.

amat returns a matrix of 0/1 integer values.

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

### Author(s)

Marco Scutari

### References

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

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

**Examples**

```

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

## the Markov blanket of A.
mb(res, "A")
## the neighbourhood of F.
nbr(res, "F")
## the arcs in the graph.
arcs(res)
## the nodes of the graph.
nodes(res)
## the adjacency matrix for the nodes of the graph.
amat(res)
## the parents of D.
parents(res, "D")
## the children of A.
children(res, "A")
## the root nodes of the graph.
root.nodes(res)
## the leaf nodes of the graph.
leaf.nodes(res)
## number of parameters of the Bayesian network.
res = set.arc(res, "A", "B")
nparams(res, learning.test)

```

---

model string utilities

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

---

**Description**

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

**Usage**

```

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

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

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

```

**Arguments**

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

**Details**

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

**Value**

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

**Author(s)**

Marco Scutari

**Examples**

```
data(learning.test)
res = set.arc(gs(learning.test), "A", "B")
res
modelstring(res)
res2 = model2network(modelstring(res))
res2
all.equal(res, res2)
```

---

naive.bayes

*Naive Bayes classifiers*

---

**Description**

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



**Usage**

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

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

**Arguments**

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

**Details**

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

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

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

**Value**

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

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

**Note**

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

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

tan.tree support whitelisting and blacklisting arcs but not their directions. Moreover it is not possible to whitelist or blacklist arcs incident on training.

predict accepts either a bn or a bn.fit object as its first argument. For the former, the parameters of the network are fitted on data, that is, the observations whose class labels the function is trying to predict.

**Author(s)**

Marco Scutari

**References**

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

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

**Examples**

```
data(learning.test)
# this is an in-sample prediction with naive Bayes (parameter learning
# is performed implicitly during the prediction).
bn = naive.bayes(learning.test, "A")
pred = predict(bn, learning.test)
table(pred, learning.test[, "A"])

# this is an in-sample prediction with TAN (parameter learning is
# performed explicitly with bn.fit).
tan = tree.bayes(learning.test, "A")
fitted = bn.fit(tan, learning.test, method = "bayes")
pred = predict(fitted, learning.test)
table(pred, learning.test[, "A"])

# this is an out-of-sample prediction, from a training test to a separate
# test set.
```

```
training.set = learning.test[1:4000, ]
test.set = learning.test[4001:5000, ]
bn = naive.bayes(training.set, "A")
fitted = bn.fit(bn, training.set)
pred = predict(fitted, test.set)
table(pred, test.set[, "A"])
```

---

node ordering utilities

*Utilities dealing with partial node orderings*

---

## Description

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

## Usage

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

## Arguments

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

## Details

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

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

## Value

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

`ordering2blacklist` and `tiers2blacklist` return 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)
ntests(res)
res = set.arc(res, "A", "B")
ord = node.ordering(res)
ord

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

tiers2blacklist(list(LETTERS[1:3], LETTERS[4:6]))
```

---

parallel integration *bnlearn - snow/parallel package integration*

---

**Description**

How to use the **bnlearn** package with the parallel computing environment provided by the **snw** or **parallel** packages.

**Parallel computing for constraint-based algorithms**

```
# load parallel and bnlearn and rsprng.
> library(parallel)
> library(bnlearn)
> cl = makeCluster(2)
# check it works.
> clusterEvalQ(cl, runif(10))
[[1]]
 [1] 0.9245585 0.1876445 0.3371175 0.2267916 0.0392876 0.9085125
 [7] 0.9041345 0.7408525 0.1537343 0.9503611

[[2]]
 [1] 0.1932651 0.8218854 0.6087155 0.9037118 0.5257906 0.8737284
 [7] 0.5225114 0.8149691 0.1671706 0.6883363
# load the data.
> data(learning.test)
# call a learning function passing the cluster object (the
```

```

# return value of the previous makeCluster() call) as a
# parameter.
> res = gs(learning.test, cluster = cl)
# note that the number of test is evenly divided between
# the two nodes of the cluster.
> clusterEvalQ(cl, test.counter())
[[1]]
[1] 47

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

```

**Author(s)**

Marco Scutari

---

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 of the y-axis.
xlim	a numeric vector with two components containing the range of the 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)
```

---

plot.bn.strength

*Plot arc strengths derived from bootstrap*

---

**Description**

Plot arc strengths derived from bootstrap resampling.

**Usage**

```
## S3 method for class 'bn.strength'
plot(x, draw.threshold = TRUE, main = NULL,
     xlab = "arc strengths", ylab = "CDF(arc strengths)", ...)
```

**Arguments**

`x` an object of class `bn.strength`.  
`draw.threshold` a boolean value. If TRUE, a dashed vertical line is drawn at the threshold.  
`main,xlab,ylab` character strings, the main title and the axes labels.  
`...` other graphical parameters.

**Note**

The `xlim` and `ylim` graphical parameters are always overridden.

**Author(s)**

Marco Scutari

**Examples**

```
data(learning.test)

start = random.graph(nodes = names(learning.test), num = 50)
netlist = lapply(start, function(net) {
  hc(learning.test, score = "bde", iss = 10, start = net) })
arcs = custom.strength(netlist, nodes = names(learning.test), cpdag = FALSE)
plot(arcs)
```

---

```
preprocess
```

---

*Pre-process data to better learn Bayesian networks*

---

**Description**

Screen and transform the data to make them more suitable for structure and parameter learning.

**Usage**

```
# discretize continuous data into factors.
discretize(data, method, breaks = 3, ordered = FALSE, ..., debug = FALSE)
# screen continuous data for highly correlated pairs of variables.
dedup(data, threshold, debug = FALSE)
```

**Arguments**

`data` a data frame containing numeric columns (for `dedup`) or a combination of numeric or factor columns (for ).  
`threshold` a numeric value between zero and one, the absolute correlation used a threshold in screening highly correlated pairs.  
`method` a character string, either `interval` for *interval discretization*, `quantile` for *quantile discretization* (the default) or `hartemink` for *Hartemink's pairwise mutual information method*.

breaks	if method is set to hartemink, an integer number, the number of levels the variables are to be discretized into. Otherwise, a vector of integer numbers, one for each column of the data set, specifying the number of levels for each variable.
ordered	a boolean value. If TRUE the discretized variables are returned as ordered factors instead of unordered ones.
...	additional tuning parameters, see below.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

### Details

discretize takes a data frame of continuous variables as its first argument and returns a second data frame of discrete variables, transformed using of three methods: interval, quantile or hartemink.

dedup screens the data for pairs of highly correlated variables, and discards one in each pair.

### Value

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

dedup returns a data frame with a subset of the columns of data.

### Note

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

### Author(s)

Marco Scutari

### References

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

### Examples

```
data(gaussian.test)
d = discretize(gaussian.test, method = 'hartemink', breaks = 4, ibreaks = 20)
plot(hc(d))
d2 = dedup(gaussian.test)
```



---

rbn *Simulate random data from a given Bayesian network*

---

### Description

Simulate random data from a given Bayesian network.

### Usage

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

### Arguments

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

### Value

A data frame with the same structure (column names and data types) of the `data` parameter (if `x` is an object of class `bn`) or with the same structure as the data originally used to fit the parameters of the Bayesian network (if `x` is an object of class `bn.fit`).

### Author(s)

Marco Scutari

### References

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

### See Also

[bn.boot](#), [bn.cv](#).

**Examples**

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

---

relevant

*Identify Relevant Nodes Without Learning the Bayesian network*


---

**Description**

Identify all the nodes relevant to compute all the conditional probability distributions for a given set of nodes.

**Usage**

```
relevant(target, context, data, test, alpha, B, debug = FALSE)
```

**Arguments**

target	a vector of character strings, the labels of nodes whose conditional probability distributions are of interest.
context	a vector of character strings, the labels of nodes on which to condition the independence tests.
data	a data frame containing either numeric or factor columns.
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 categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> for details.
alpha	a numeric value, the target nominal type I error rate. If none is specified, the default value is 0.05.
B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the test argument is not a permutation test.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

**Value**

relevant returns a vector of character strings, the labels of the relevant nodes.

**Note**

This algorithm selects all the nodes that are relevant at all, not only those that are significantly so. Therefore, to be discarded a node must be completely unrelated to any of the target nodes, not just weakly dependent. On the good side, relevant nodes are correctly identified even for data sets whose probability structure is not faithful to any directed acyclic graph.

**Author(s)**

Marco Scutari

**References**

Pena JM, Nilsson R, Björkegren J, Tegner J (2006). "Identifying the Relevant Nodes Without Learning the Model". In "Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI2006)", pp. 367-374.

**Examples**

```
data(learning.test)
X = as.factor(sample(c("x1", "x2"), nrow(learning.test), replace = TRUE))
relevant("A", data = cbind(learning.test, X))
relevant("A", context = "B", data = learning.test,)
```

---

score

*Score of the Bayesian network*

---

**Description**

Compute the score of the Bayesian network.

**Usage**

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

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

**Arguments**

x, object	an object of class bn.
data	a data frame containing the data the Bayesian network was learned from.
type	a character string, the label of a network score. If none is specified, the default score is the <i>Bayesian Information Criterion</i> for both discrete and continuous data sets. See <a href="#">bnlearn-package</a> for details.

debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.
...	extra arguments from the generic method (for the AIC and logLik functions, currently ignored) or additional tuning parameters (for the score function).
k	a numeric value, the penalty per parameter to be used; the default $k = 1$ gives the expression used to compute the AIC in the context of scoring Bayesian networks.

## Details

Additional parameters of the score function:

- `iss`: the imaginary sample size, used by the Bayesian Dirichlet equivalent score (both the `bde` and `mbde`) and the Bayesian Gaussian score (`bge`). It is also known as “equivalent sample size”. The default value is equal to 10 for both the `bde/mbde` scores and `bge`.
- `exp`: a list of indexes of experimental observations (those that have been artificially manipulated). Each element of the list must be named after one of the nodes, and must contain a numeric vector with indexes of the observations whose value has been manipulated for that node.
- `k`: the penalty per parameter to be used by the AIC and BIC scores. The default value is 1 for AIC and  $\log(\text{nrow}(\text{data}))/2$  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.)
- `prior`: the prior distribution to be used with the Bayesian Dirichlet equivalent score (`bde`) and the Bayesian Gaussian score (`bge`). Possible values are `uniform` (the default), `vsp` (the Bayesian variable selection prior, which puts a probability of inclusion on parents) and `cs` (the Castelo & Siebes prior, which puts an independent prior probability on each arc and direction).
- `beta`: the parameter associated with `prior`. If `prior` is `uniform`, `beta` is ignored. If `prior` is `vsp`, `beta` is the probability of inclusion of an additional parent (the default is  $1/\text{ncol}(\text{data})$ ). If `prior` is `cs`, `beta` is a data frame with columns `from`, `to` and `prob` specifying the prior probability for a set of arcs. A uniform probability distribution is assumed for the remaining arcs.

## Value

A numeric value, the score of the Bayesian network.

## Note

AIC and BIC are computed as  $\text{logLik}(x) - k * \text{nparams}(x)$ , that is, the classic definition rescaled by -2. Therefore higher values are better, and for large sample sizes BIC converges to  $\log(\text{BDe})$ .

When using the Castelo & Siebes prior in structure learning, the probability of not including an arc is bounded away from zero by adding  $2 * \text{.Machine}\$double.\text{eps}$  and subtracting  $\text{.Machine}\$double.\text{eps}$  from each of the probabilities of the possible arc directions. This dramatically improves structure learning, which is less likely to get stuck when starting from an empty graph. As an alternative to prior probabilities, a blacklist can be used to prevent arc from being included in the network. `beta`

is not modified when the prior is used from functions other than those implementing score-based and hybrid structure learning.

### Author(s)

Marco Scutari

### References

Castelo R, Siebes A (2000). "Priors on Network Structures. Biasing the Search for Bayesian Networks". *International Journal of Approximate Reasoning*, **24**(1), 39-57.

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.

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

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

## BDe with a prior.
beta = data.frame(from = c("A", "D"), to = c("B", "F"),
                  prob = c(0.2, 0.5), stringsAsFactors = FALSE)
score(res, learning.test, type = "bde", prior = "cs", beta = beta)

## k2 score on the other hand is not score equivalent.
score(res, learning.test, type = "k2")
score(res2, learning.test, type = "k2")

## equivalent to logLik(res, learning.test)
score(res, learning.test, type = "loglik")

## equivalent to AIC(res, learning.test)
```

```
score(res, learning.test, type = "aic")
```

---

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, maxp = Inf, optimized = TRUE)
tabu(x, start = NULL, whitelist = NULL, blacklist = NULL, score = NULL, ...,
     debug = FALSE, tabu = 10, max.tabu = tabu, max.iter = Inf, maxp = 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 <a href="#">bnlearn-package</a> for details.
<code>...</code>	additional tuning parameters for the network score. See <a href="#">score</a> for details.
<code>debug</code>	a boolean value. If TRUE 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 tabu 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>maxp</code>	the maximum number of parents for a node. The default value is <code>Inf</code> .
<code>optimized</code>	a boolean value. See <a href="#">bnlearn-package</a> for details.

**Value**

An object of class bn. See [bn-class](#) for details.

**Author(s)**

Marco Scutari

**References**

- Russell SJ, Norvig P (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.
- Korb K, Nicholson AE (2010). *Bayesian Artificial Intelligence*. Chapman & Hall/CRC, 2nd edition.
- Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-03-153.
- Daly R, Shen Q (2007). "Methods to Accelerate the Learning of Bayesian Network Structures". In "Proceedings of the 2007 UK Workshop on Computational Intelligence", Imperial College, London.

**See Also**

[constraint-based algorithms](#), [hybrid algorithms](#),  
[local discovery algorithms](#).

---

single-node local discovery

*Discover the structure around a single node*

---

**Description**

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

**Usage**

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

**Arguments**

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

blacklist	a vector of character strings, the labels of the blacklisted nodes.
start	a vector of character strings, the labels of the nodes to be included in the Markov blanket before the learning process (in <code>learn.mb</code> ). Note that the nodes in <code>start</code> can be removed from the Markov blanket by the learning algorithm, unlike the nodes included due to whitelisting.
test	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the <i>mutual information</i> for categorical variables, the Jonckheere-Terpstra test for ordered factors and the <i>linear correlation</i> for continuous variables. See <a href="#">bnlearn-package</a> for details.
alpha	a numeric value, the target nominal type I error rate.
B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the <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.

**Value**

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

**Author(s)**

Marco Scutari

**See Also**

[constraint-based algorithms](#), [local discovery algorithms](#).

---

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 <a href="#">graphviz.plot</a> for details.
layout	a character string, the layout parameter to be passed to <b>Rgraphviz</b> . Possible values are dots, neato, twopi, circo and fdp. See <b>Rgraphviz</b> documentation for details.
shape	a character string, the shape of the nodes. Can be circle, ellipse or rectangle.
main	a character string, the main title of the graph. It's plotted at the top of the graph.
sub	a character string, a subtitle which is plotted at the bottom of the graph.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

**Details**

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

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

Non-significant arcs are plotted as dashed lines.

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

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

if the coefficients are computed from conditional independence tests, by

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

for bootstrap estimates or by the quantiles

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

of the significant coefficients if network scores are used.

**Value**

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

**Author(s)**

Marco Scutari

**Examples**

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

## End(Not run)
```

---

test counter

*Manipulating the test counter*

---

**Description**

Check, increment or reset the test/score counter used in structure learning algorithms.

**Usage**

```
test.counter()
increment.test.counter(i = 1)
reset.test.counter()
```

**Arguments**

*i* a numeric value, which is added to the test counter.

**Value**

A numeric value, the current value of the test counter.

**Author(s)**

Marco Scutari

**Examples**

```
data(learning.test)
hc(learning.test)
test.counter()
reset.test.counter()
test.counter()
```

# Index

- \*Topic **IO**
  - foreign files utilities, 48
- \*Topic **classes**
  - bn class, 17
  - bn.fit class, 26
  - bn.kcv class, 31
  - bn.strength class, 32
- \*Topic **datasets**
  - alarm, 9
  - asia, 16
  - clgaussian.test, 36
  - coronary, 41
  - gaussian.test, 50
  - hailfinder, 57
  - insurance, 62
  - learning.test, 64
  - lizards, 65
  - marks, 67
- \*Topic **documentation**
  - deal integration, 46
  - parallel integration, 76
- \*Topic **file**
  - foreign files utilities, 48
- \*Topic **graphs**
  - arc operations, 11
  - bn.fit utilities, 28
  - compare, 37
  - constraint-based algorithms, 39
  - cpdag, 42
  - dsep, 47
  - graph generation utilities, 51
  - graph utilities, 54
  - hybrid algorithms, 60
  - local discovery algorithms, 66
  - misc utilities, 68
  - model string utilities, 71
  - score-based algorithms, 86
  - single-node local discovery, 87
- \*Topic **hplot**
  - bn.fit plots, 27
  - graphviz.plot, 55
  - plot.bn, 77
  - plot.bn.strength, 78
  - strength.plot, 88
- \*Topic **htest**
  - arc.strength, 13
  - choose.direction, 33
  - ci.test, 34
  - score, 83
- \*Topic **manip**
  - configs, 38
  - preprocess, 79
- \*Topic **models**
  - constraint-based algorithms, 39
  - hybrid algorithms, 60
  - local discovery algorithms, 66
  - relevant, 82
  - score-based algorithms, 86
  - single-node local discovery, 87
- \*Topic **multivariate**
  - bn.boot, 18
  - bn.cv, 20
  - bn.fit, 23
  - constraint-based algorithms, 39
  - cpdag, 42
  - cpquery, 43
  - dsep, 47
  - gRain integration, 50
  - graph integration, 53
  - hybrid algorithms, 60
  - local discovery algorithms, 66
  - naive.bayes, 72
  - node ordering utilities, 75
  - rbn, 81
  - relevant, 82
  - score-based algorithms, 86
  - single-node local discovery, 87
- \*Topic **nonparametric**

- bn.boot, 18
- bn.cv, 20
- \*Topic **package**
  - bnlearn-package, 3
- \*Topic **utilities**
  - arc operations, 11
  - bn.fit utilities, 28
  - foreign files utilities, 48
  - gRain integration, 50
  - graph generation utilities, 51
  - graph integration, 53
  - graph utilities, 54
  - misc utilities, 68
  - model string utilities, 71
  - node ordering utilities, 75
  - rbn, 81
  - test counter, 90
- \$<-bn.fit (bn.fit), 23
- acyclic (graph utilities), 54
- AIC.bn (score), 83
- AIC.bn.fit (bn.fit utilities), 28
- alarm, 9
- all.equal.bn (compare), 37
- amat (misc utilities), 68
- amat<- (misc utilities), 68
- aracne, 5
- aracne (local discovery algorithms), 66
- arc operations, 11
- arc.strength, 13, 32, 33, 35, 85
- arcs (misc utilities), 68
- arcs<- (misc utilities), 68
- as.bn (model string utilities), 71
- as.bn.fit (gRain integration), 50
- as.bn.graphAM (graph integration), 53
- as.bn.graphNEL (graph integration), 53
- as.character.bn (model string utilities), 71
- as.grain (gRain integration), 50
- as.graphAM (graph integration), 53
- as.graphNEL (graph integration), 53
- asia, 16
- averaged.network, 13
- averaged.network (arc.strength), 13
- BIC.bn (score), 83
- BIC.bn.fit (bn.fit utilities), 28
- blacklist (misc utilities), 68
- bn class, 17
- bn-class (bn class), 17
- bn.boot, 18, 22, 81
- bn.cv, 20, 20, 81
- bn.fit, 21, 23, 28, 31, 81
- bn.fit class, 26
- bn.fit plots, 27
- bn.fit utilities, 28
- bn.fit-class (bn.fit class), 26
- bn.fit.barchart (bn.fit plots), 27
- bn.fit.dnode (bn.fit class), 26
- bn.fit.dotplot (bn.fit plots), 27
- bn.fit.gnode (bn.fit class), 26
- bn.fit.histogram (bn.fit plots), 27
- bn.fit.qqplot (bn.fit plots), 27
- bn.fit.xyplot (bn.fit plots), 27
- bn.kcv class, 31
- bn.kcv-class (bn.kcv class), 31
- bn.kcv.list class (bn.kcv class), 31
- bn.kcv.list-class (bn.kcv class), 31
- bn.net (bn.fit), 23
- bn.strength (bn.strength class), 32
- bn.strength class, 32
- bn.strength-class (bn.strength class), 32
- bnlearn (bnlearn-package), 3
- bnlearn-package, 3
- boot.strength, 14, 32
- boot.strength (arc.strength), 13
- cextend (cpdag), 42
- children (misc utilities), 68
- children<- (misc utilities), 68
- choose.direction, 15, 33, 35, 85
- chow.liu, 5
- chow.liu (local discovery algorithms), 66
- ci.test, 15, 34
- clgaussian.test, 36
- coef.bn.fit (bn.fit utilities), 28
- compare, 37
- compelled.arcs (misc utilities), 68
- configs, 38
- constraint-based algorithms, 39, 62, 67, 87, 88
- coronary, 41
- cpdag, 42
- cpdist (cpquery), 43
- cpquery, 43
- custom.fit (bn.fit), 23

- custom.strength, [14](#), [32](#)
- custom.strength (arc.strength), [13](#)
- deal integration, [46](#)
- dedup (preprocess), [79](#)
- degree (misc utilities), [68](#)
- degree, bn-method (misc utilities), [68](#)
- degree, bn.fit-method (misc utilities), [68](#)
- degree, bn.naive-method (misc utilities), [68](#)
- degree, bn.tan-method (misc utilities), [68](#)
- directed (graph utilities), [54](#)
- directed.arcs (misc utilities), [68](#)
- discretize (preprocess), [79](#)
- drop.arc (arc operations), [11](#)
- drop.edge (arc operations), [11](#)
- dsep, [47](#)
- empty.graph (graph generation utilities), [51](#)
- fast.iamb, [4](#)
- fast.iamb (constraint-based algorithms), [39](#)
- fitted.bn.fit (bn.fit utilities), [28](#)
- foreign files utilities, [48](#)
- gaussian.test, [50](#)
- gRain integration, [50](#)
- graph generation utilities, [51](#)
- graph integration, [53](#)
- graph utilities, [54](#)
- graphviz.plot, [55](#), [78](#), [89](#)
- gs, [3](#)
- gs (constraint-based algorithms), [39](#)
- hailfinder, [57](#)
- hamming (compare), [37](#)
- hc, [4](#)
- hc (score-based algorithms), [86](#)
- hybrid algorithms, [40](#), [60](#), [67](#), [87](#)
- iamb, [4](#)
- iamb (constraint-based algorithms), [39](#)
- in.degree (misc utilities), [68](#)
- incident.arcs (misc utilities), [68](#)
- incoming.arcs (misc utilities), [68](#)
- increment.test.counter (test counter), [90](#)
- insurance, [62](#)
- inter.iamb, [4](#)
- inter.iamb (constraint-based algorithms), [39](#)
- leaf.nodes (misc utilities), [68](#)
- learn.mb (single-node local discovery), [87](#)
- learn.nbr (single-node local discovery), [87](#)
- learning.test, [64](#)
- lizards, [65](#)
- local discovery algorithms, [40](#), [62](#), [66](#), [87](#), [88](#)
- logLik.bn (score), [83](#)
- logLik.bn.fit (bn.fit utilities), [28](#)
- marks, [67](#)
- mb (misc utilities), [68](#)
- misc utilities, [68](#)
- mmhc, [4](#)
- mmhc (hybrid algorithms), [60](#)
- mmpc, [4](#)
- mmpc (local discovery algorithms), [66](#)
- model string utilities, [71](#)
- model2network (model string utilities), [71](#)
- modelstring (model string utilities), [71](#)
- modelstring<- (model string utilities), [71](#)
- moral (cpdag), [42](#)
- mutilated (cpquery), [43](#)
- naive.bayes, [5](#), [72](#)
- narcs (misc utilities), [68](#)
- nbr (misc utilities), [68](#)
- nnodes (misc utilities), [68](#)
- node ordering utilities, [75](#)
- node.ordering (node ordering utilities), [75](#)
- nodes (misc utilities), [68](#)
- nodes, bn-method (misc utilities), [68](#)
- nodes, bn.fit-method (misc utilities), [68](#)
- nodes, bn.naive-method (misc utilities), [68](#)
- nodes, bn.tan-method (misc utilities), [68](#)
- nodes<- (misc utilities), [68](#)

- nodes<- ,bn-method (misc utilities), 68
- nodes<- ,bn.fit-method (misc utilities), 68
- nodes<- ,bn.naive-method (misc utilities), 68
- nodes<- ,bn.tan-method (misc utilities), 68
- nparams (misc utilities), 68
- ntests (misc utilities), 68
  
- ordering2blacklist (node ordering utilities), 75
- out.degree (misc utilities), 68
- outgoing.arcs (misc utilities), 68
  
- parallel integration, 4, 76
- parents (misc utilities), 68
- parents<- (misc utilities), 68
- path (graph utilities), 54
- pdag2dag, 24
- pdag2dag (graph utilities), 54
- plot.bn, 56, 77
- plot.bn.kcv (bn.cv), 20
- plot.bn.strength, 78
- predict.bn.fit (bn.fit utilities), 28
- predict.bn.naive (naive.bayes), 72
- predict.bn.tan (naive.bayes), 72
- preprocess, 79
  
- random.graph (graph generation utilities), 51
- rbn, 20, 22, 81
- read.bif (foreign files utilities), 48
- read.dsc (foreign files utilities), 48
- read.net (foreign files utilities), 48
- relevant, 82
- reset.test.counter (test counter), 90
- residuals.bn.fit (bn.fit utilities), 28
- reverse.arc (arc operations), 11
- reversible.arcs (misc utilities), 68
- root.nodes (misc utilities), 68
- rsmax2, 4
- rsmax2 (hybrid algorithms), 60
  
- score, 14, 15, 24, 33, 61, 83, 86
- score-based algorithms, 40, 62, 67, 86
- set.arc, 24
- set.arc (arc operations), 11
- set.edge (arc operations), 11
  
- shd (compare), 37
- si.hiton.pc, 5
- si.hiton.pc (local discovery algorithms), 66
- sigma (bn.fit utilities), 28
- single-node local discovery, 87
- skeleton (graph utilities), 54
- strength.plot, 15, 32, 88
- subgraph (graph utilities), 54
  
- tabu, 4
- tabu (score-based algorithms), 86
- test counter, 90
- test.counter (test counter), 90
- tiers2blacklist (node ordering utilities), 75
- tree.bayes, 5
- tree.bayes (naive.bayes), 72
  
- undirected.arcs (misc utilities), 68
  
- vstructs (cpdag), 42
  
- whitelist (misc utilities), 68
- write.bif (foreign files utilities), 48
- write.dot (foreign files utilities), 48
- write.dsc (foreign files utilities), 48
- write.net (foreign files utilities), 48